Execution Contexts For Determining Trust In A Higher-Order π-Calculus

Mark Hepburn [1], David Wright [2]

School of Computing
University of Tasmania
GPO Box 252-100
Tasmania, 7001
Australia

Abstract

A type system for the higher order π-calculus is presented, using boolean annotations to carry information about the integrity of data. This extends previous work by the authors for a first order calculus; we argue that in a higher order setting certain notions of trustedness depend on the potential for communication during execution. In addition, we develop a statically determinable method for performing safe run-time trust coercions.

1 Introduction

Systems using mobile code have the potential to revolutionise the way we access computing resources and information in general. This is because of the tangible benefits such systems offer: the computing resources can be spread across multiple locations; clients accessing remotely hosted applications receive better performance by down-loading applets to perform some of the tasks that would ordinarily have taken several transactions with the server to complete; mobile agents can be transmitted to perform searching work of various kinds; access to information need no longer be restricted to the machine it is stored on; and so on.

These benefits do not come without corresponding risk of course; before running code from a remote source on your machine one would be well advised to make certain it can do no harm, so it is no surprise that research into ways of detecting, preventing and mitigating such dangers is flourishing [3,12,17,18,23,24,25]. Some approaches focus on the code itself (such as

[1] Email: Mark.Hepburn@utas.edu.au
[2] Email: David.Wright@utas.edu.au

An extended abstract of this report has been submitted to the Foundations of Global Computing workshop, 2003.

proof-carrying code [12] and java byte-code verification [25]); others take a sand-box approach, mitigating the damage possible from untrusted code (for example, the box-π calculus [17,18] and the java sand-box [25]); while still others take the approach of examining the interactions between processes (for example, protocol analysis [1,2], trust-based navigation [23,24] and the lattice-based flow analysis approach [3]).

Many of these approaches utilise a types-based analysis; the benefits of a strong typing system in preventing run-time errors are well known, so it is perhaps natural to extend the type system to carry additional information about the program such as its security level.

Of these though, most suffer from a lack of flexibility. As an example, consider the case of data plus a signature (assuming that the signature algorithm is unforgeable) sent along a potentially malicious channel. Due to the danger of tampering a typical static system would mark all data as untrusted; however in effect it has *variable* trustedness, because we now have the ability to, on a case by case basis, determine the trustedness of each piece of data at run-time.

An interesting approach to this problem was proposed by Ørbæk and Palsberg [13,14] allowing the programmer to specify coercions on the trustedness of data, for example casting a variable as trusted after appropriate checks have been made to determine that that was in fact the case. Three operators lie at the heart of their system; two coercion operators (`trust` and `distrust`), and a third, `check`, that is used to verify the trustedness of its argument. The intention is that the programmer judiciously applies these operators at suitable points in the program, then the compiler can calculate — statically — if the program is safe with respect to data trustedness. An important result is that if the annotated program is safe, then so is the same program stripped of the three additional operators. While this is undoubtedly important research, it still relies on the programmer to correctly use the operators (in that *any* data is considered trusted if so coerced); in effect a program can be determined safe, but only modulo correct use of trust coercion by the programmer ("With great power comes great responsibility"!).

Seeking to address this issue, the authors introduced a system [4] in which trust coercion is still available to the programmer, but its indiscriminate usage is not: the direction of coercion (either trusted or untrusted) is determined solely by the results of run-time verification procedures, and the execution path also branches based on these results. In this manner the programmer retains the flexibility of being able to use coercion, but loses the responsibility of getting it right.

That work (which also differed from the functional [14] and imperative implementations [13] of Ørbæk and Palsberg by considering a distributed system) used a version of the first order π-calculus for its analysis. Because of this only atomic data could be considered; this is both restrictive and unrealistic: as mentioned earlier, one of the main attractions of distributed computing is

allowing entire programs (agents) to be transmitted. This paper addresses this issue by examining a higher-order modeling calculus; the additional flexibility allows us to consider the ramifications of potentially malicious agents moving through a network and running in different environments, and due to the dynamic topology represented by the π-calculus the possibility of new connections being opened with unknown processes.

1.1 Examples and Motivation

We take a slightly different focus from other similar systems (commonly referred to as *flow analysis*), in that rather than examining programs to detect data leakage (for example, data with a high security rating being exposed to processes with a lower security clearance) we take the view that in many situations this is unavoidable: much of the internet for example is potentially exposed to the viewing and tampering of data. We accept this, and merely wish to avoid using data that cannot be verified or trusted in any secure computation. Elsewhere [4] this was achieved by making the trustedness of the data received along a channel be related to the trustedness of the channel itself, so all data from an untrusted channel is likewise untrusted (the converse is not true; we still permit untrusted data to be transmitted along trusted channels).

In a higher-order situation things get a little more complicated. Because processes themselves can be transmitted, potentially along channels that expose them to tampering, this implies that they too can be assigned a trustedness value. Perhaps a logical extension of this assumption is that, just as data from an untrusted channel is untrusted, so is data from an untrusted process. However, it is still not that easy: we must consider what it means for a process to be untrusted. For example, if an agent whose trustworthiness cannot be guaranteed is run within a sand-box so that it is incapable of harming the system, should it still be considered dangerous? We take the view that it should not; that the final trustedness "rating" of an agent depends on its potential to inflict harm or corrupt other agents in the environment in which it is run.

To achieve this in a type system we introduce the concept of *contexts*; constructs that carry information about the manner in which different parts of a process are liable to be used. Every deduction for a program in our system ends in an application of a rule called *final*; this takes the information embedded in the contexts and computes a final — environment dependent — trust value. The information concerning the execution environment (for example, the access ports in a sand-box) is also carried through a deduction expressed as the external context. It should be noted that to recompute the trustedness of an agent in a different setting currently the entire deduction must be repeated; this is certainly inefficient and will hopefully be improved upon in later work.

1.2 Layout

The report is presented as follows. We first introduce the higher order calculus we use in our modeling in section 2, then in section 3 describe a syntactic extension to the language that forms the heart of our system. A type system is described in section 4. An inference algorithm for the type system is presented in section 5, then the safety of the system and the correctness of the algorithm follows in section 6. We conclude in section 7.

No previous experience with the π-calculus is assumed, however at least a passing familiarity would be an advantage.

2 Preliminaries

The π-calculus [10,11,9], based on CCS [8], was proposed by Milner, Parrow and Walker as a way of reasoning about concurrency in a similar way that the λ-calculus enables reasoning about computation. Reduction occurs as a result of communication of data along channels; at its purest, all data is simply channels and processes are made up of combinations of channels.

The higher-order π-calculus was conceived as a logical extension in which processes as well as channels could be transmitted along channels. Sangiorgi [16] demonstrated that the higher-order calculus could satisfactorily be encoded in the first-order calculus, and that a similar mapping also preserved typing information. [3] Nevertheless, the expression of higher order concepts directly exposes the issues of security in a system with mobile processes and provides a suitable environment for developing our ideas.

2.1 Syntax

We now consider the syntax formally, with a brief informal discussion of the main constructs. Two processes executing in parallel, respectively denoted P and Q, is written as P|Q. The construct P + Q denotes a choice of execution between P or Q (but not both); this is a non-deterministic choice, but in practice the outcome is usually decided by whichever has the prior opportunity to reduce.

The main construct is that of communication, which occurs along named channels. A process that sends the single name y along a channel x then continues as the process P is written as $\overline{x}[y]$.P; note that the name being sent is enclosed in square brackets and for additional clarity the output channel is written with a bar over the top to distinguish it from an identically named input channel. The dual of this construct is the process that receives, along channel x, a variable which it binds to the name z in the process Q and is written as $x(z)$.Q. We note that most versions of the π-calculus — including

[3] Actually, *sorting*; see section 2.3. The corresponding verification that the mapping preserved typing structure was presented by Vasconcelos [20].

that used here — are *polyadic*; that is tuples of names are also allowed to be transmitted. We will use the terms channel, data, and name interchangeably.

Let names be ranged over by x, y, z, and process variables by X and Y. K refers to either a name or a process, and U and V to either a name or a variable. The syntax we use is based on Vasconcelos' [20] and is shown in definition 2.1:

Definition 2.1

$$\mathrm{P} ::= \mathbf{0} \mid !\mathrm{P} \mid \mathrm{P}|\mathrm{Q} \mid \mathrm{P} + \mathrm{Q} \mid (\nu U)\mathrm{P} \mid X \mid x(\vec{U}).\mathrm{P} \mid \overline{x}[\vec{K}].\mathrm{P}$$

The remaining constructs require just a little explanation; $\mathbf{0}$ is the null process that does nothing, and the *restriction* operator $(\nu U)\mathrm{P}$ creates a unique name (resp. variable) with scope P. The *replication* construct !P represents an infinite source of P and can be defined inductively as $!\mathrm{P} \triangleq \mathrm{P}|!\mathrm{P}$. The reduction rules (see section 2.2) guarantee that it does not replicate infinitely. It should be noted that the syntax used in this material lacks constructs usually present in other presentations; this is for reasons of simplicity, and an (informal) justification is given in section 2.2.1.

2.2 Semantics

The semantics of the calculus are given in terms of a reduction semantics; in the core calculus [9] there is a single reduction axiom, and three inference rules on the axiom.

Firstly, let us define structural congruence on processes: this is fairly trivial; letting $\mathcal{P}$ be the set of all processes then $\{\mathcal{P}, \equiv, |, \mathbf{0}\}$ forms an Abelian monoid, and so does $\{\mathcal{P}, \equiv, +, \mathbf{0}\}$. Processes are structurally congruent if they are identical up to the renaming of bound names, and we also comment on the scope of restrictions and replications. More formally:

Definition 2.2

$$\begin{array}{ll} \mathrm{P}|\mathrm{Q} \equiv \mathrm{Q}|\mathrm{P} & \mathrm{P} + \mathrm{Q} \equiv \mathrm{Q} + \mathrm{P} \\ \mathrm{P}|\mathbf{0} \equiv \mathrm{P} & \mathrm{P} + \mathbf{0} \equiv \mathrm{P} \\ \mathrm{P}|(\mathrm{Q}|\mathrm{R}) \equiv (\mathrm{P}|\mathrm{Q})|\mathrm{R} & \mathrm{P} + (\mathrm{Q} + \mathrm{R}) \equiv (\mathrm{P} + \mathrm{Q}) + \mathrm{R} \\ \\ !\mathrm{P} \equiv \mathrm{P}|!\mathrm{P} & (\nu U)\mathrm{P}|\mathrm{Q} \equiv (\nu U)(\mathrm{P}|\mathrm{Q}) \quad U \notin FV(\mathrm{Q}) \end{array}$$

Now we can define our reduction semantics. The core axiom in the standard calculus is that of communication:

$$(\ldots + \overline{x}[\vec{K}].\mathrm{P})|(x(\vec{U}).\mathrm{Q} + \ldots) \rightarrow \mathrm{P}|\mathrm{Q}\{\vec{K}/\vec{U}\}$$

The three inference rules on this axiom can be summarised as: reduction may occur under a restriction (but *not* under a communication prefix) or a parallel composition, and structurally congruent terms have the same reduc-

tions:

$$\frac{\mathrm{P} \to \mathrm{P}'}{\mathrm{P}|\mathrm{Q} \to \mathrm{P}'|\mathrm{Q}} \qquad \frac{\mathrm{P} \to \mathrm{P}'}{(\nu U)\mathrm{P} \to (\nu U)\mathrm{P}'} \qquad \frac{\mathrm{Q} \equiv \mathrm{P} \quad \mathrm{P} \to \mathrm{P}' \quad \mathrm{P}' \equiv \mathrm{Q}'}{\mathrm{Q} \to \mathrm{Q}'}$$

Note that they also do *not* allow reduction under replication; the replication must be expanded out as many times as necessary then the reduction performed on the individual processes.

Definition 2.3 (Labeled Reductions) *Occasionally it is useful to specify the precise step involved in a reduction; basically, the name(s) communicated. Given the reduction*

$$\overline{x}[\vec{K}].\mathrm{P}|x(\vec{U}).\mathrm{Q} \to \mathrm{P}|\mathrm{Q}\{\vec{K}/\vec{U}\}$$

then we label the individual redeces as follows:

$$\overline{x}[\vec{K}].\mathrm{P} \xrightarrow{\overline{x}[\vec{K}]} \mathrm{P}$$

$$x(\vec{U}).\mathrm{Q} \xrightarrow{x(\vec{K})} \mathrm{Q}\{\vec{K}/\vec{U}\}$$

We label an internal *reduction — that is, one in which no other processes are involved — as*

$$\overline{x}[\vec{K}].\mathrm{P}|x(\vec{U}).\mathrm{Q} \xrightarrow{\tau} \mathrm{P}|\mathrm{Q}\{\vec{K}/\vec{U}\}$$

2.2.1 Abstractions and Applications

Readers familiar with other presentations of the higher-order π-calculus [16,20] may have noticed that our syntax does not admit abstractions or applications. This is an intentional omission, driven by a desire for simplicity and the belief that with some manipulation the benefits gained by admitting these constructs can still be encoded in our core calculus.

Most versions of a higher-order π-calculus include the ability to *abstract* a process on one or more names or variables (e.g. $(U)\mathrm{P}$, with the corresponding ability (*application*) to instantiate the bound names/variables with given values, usually written as $\mathrm{F}\langle K\rangle$ where F is of the form $(U)\mathrm{P}$. The usual argument is that this makes the system truly higher-order, because as well as sending processes you can send processes abstracted on arbitrary variables; we argue though that informally (that is, we do not provide a formal translation, nor do we claim that it would be trivial since as will be seen our informal encoding involves changing the arity of names) the same benefits may be achieved using our core calculus. We illustrate this claim with the following example (2.4):

Example 2.4 *consider the following fragment of code (and associated reduction) in a system with abstraction and application:*

$$\begin{aligned}\overline{x}\left[(Y)\mathrm{P}\right].\mathbf{0}|x(X).X\langle\mathrm{Q}\rangle &\to^{*} (Y)\mathrm{P}\langle\mathrm{Q}\rangle \\ &\to \mathrm{P}\{\mathrm{Q}/Y\}\end{aligned}$$

A similar reduction can be achieved by sending the process, in the form of an

input located at a fresh name instead of as an abstraction, along with the name at which it is located:

$$(\nu y)\overline{x}\left[y\left(y(Y).\mathrm{P}\right)\right].\mathbf{0}|x(zX).(\overline{z}[\mathrm{Q}].\mathbf{0}|X) \rightarrow^* (\nu y)\left(\overline{y}[\mathrm{Q}].\mathbf{0}|y(Y).\mathrm{P}\right) \rightarrow^* \mathrm{P}\{\mathrm{Q}/Y\}$$

2.3 A Type System

2.3.1 A First Notion of Types

We now sketch the type system most commonly used in the π-calculus. The primary goal of any type system is to prevent certain types of run-time errors occurring: in the polyadic π-calculus the most likely source of error is arity mismatch; for example, a process sends three names to a process expecting to receive only two. Thus our type system will enforce a policy on the number (and type) of names that channels with that type can carry.

We start then by representing the type of a channel as a list of types. It is necessary to be careful however, and distinguish between the type of names, which can themselves carry other names, and the type of processes that can merely be transmitted. To distinguish the two, we use parentheses for channel types and square brackets for process types.

If we write types as σ with variables be ranged over by α, then our type syntax becomes (definition 2.5):

Definition 2.5

$$\sigma ::= [] \mid (\vec{\sigma}) \mid \alpha$$

The astute reader will no doubt have noticed that this scheme makes no allowances for recursive types; for example, the types of channels capable of transmitting themselves. This is a deliberate omission; the added complexity (primarily in the unification algorithms used in type inference) of recursive types detracts from the main thrust of this report which is a scheme for safe run-time coercion. It is worth noting that early π-calculus researchers favoured a slightly different approach; that of a *sorting* as opposed to types. The central concept is that each name is assigned a sort (an atom), then a separate mapping (known as a *sorting*) is used to map sorts to lists of sorts that names of that sort can carry. This has two advantages over self-contained types; name-equivalence (as opposed to mere structural equivalence), and a simpler treatment of recursion (recursion does not in fact exist as a separate concept, since it is inherent in the sorting). The added complexity in the analysis induced by the sorting though seems to out-weigh any benefits, and recent researchers also appear to favour types (e.g. [20]). The interested reader is referred to previous work by the authors which described a similar system to that presented here, for the first-order calculus only, using sorts instead of types [4].

2.3.2 Adding Trustedness Information

Now to our base type structure we wish to add some additional information, representing the integrity of the data or process in question. Note that although in this paper we generally refer to "integrity" as meaning "untampered with and from a trusted source" (that is, a security measure), it could just as easily refer to, for example, the reliability of data, such as data received from a noisy channel. The system presented does not distinguish in this matter.

The approach we take in adding annotations is based on that initially proposed by Wright [22], using a boolean algebra structure. We take trustedness (T) to correspond to truth or 1, and untrustedness (U) to correspond to false or 0. The syntax of annotations, ranged over by b where variables are ranged over by i, j, is shown in definition 2.6:

Definition 2.6 (boolexp)

$$b ::= i \mid \mathrm{T} \mid \mathrm{U} \mid b + b \mid b \cdot b \mid \hat{b}$$

The operations on annotations are inductively defined as follows:

Definition 2.7

$$b \cdot \mathrm{T} = b \quad b + \mathrm{T} = \mathrm{T} \quad \hat{\mathrm{T}} = \mathrm{U}$$
$$b \cdot \mathrm{U} = \mathrm{U} \quad b + \mathrm{U} = b \quad \hat{\mathrm{U}} = \mathrm{T}$$

We will at times omit the '$\cdot$' and simply write bc where the meaning is clear. It will be seen later that the type system itself (section 4) makes no use of summation or negation, however the boolean unification algorithm which will be used in the inference algorithm (section 5) does, so for completeness we define them now.

We also extend the definition of multiplication to cover types as well, with the logical extension to cover sequences of types (definition 2.8):

Definition 2.8

$$b \cdot \sigma^c = \sigma^{bc}$$

We additionally define the following relation:

Definition 2.9 *Define $\leq$ (and, by extension, $\geq$) as being the least reflexive, associative and transitive relation over annotations satisfying the following:*

$$\mathrm{U} \leq b \leq \mathrm{T}$$

Our complete annotated type syntax can be seen in definition 2.10. Note that because we do not admit abstractions in the term syntax, the only process type is that of a well-formed process (that is, []).

Definition 2.10

$$\sigma ::= [] \mid \left(\vec{\sigma^b}\right) \mid \alpha$$

2.4 Execution Contexts

This type structure by itself is enough to guarantee several important properties, such as no data from an untrusted channel or process being used in a trusted computation. However, as hinted at in the introduction, things become more complicated in a higher order setting because we need to consider what can cause a process to be "untrusted". Some causes are (relatively) trivial; a process received from an untrusted channel for example must be considered untrusted as it may have been tampered with or replaced by a malicious agent. Likewise, in an explicitly typed system (which we don't consider here) certain processes may be flagged as untrusted by the programmer. In addition though, we take the view that a process may become corrupted if it can *communicate* with an untrusted process.

To understand the motivation behind this decision it is wise to consider a few salient examples. Consider first of all a program consisting of two processes running in parallel, one trusted and the other untrusted. What should the overall trustedness of the program be? A common reaction is to assume untrusted (certainly the safest option). However if the untrusted process is completely closed, that is it cannot communicate with anyone, then it is probably reasonable to say that under those circumstances the program could safely be considered trusted.

Consider a related case, consisting of a single malicious process that is not closed yet none-the-less has no means of interacting (and therefore harming) its environment. In this case too, it is probably safe to say that *in that context* that process can be trusted.

As a final example, consider several processes running in parallel. The first, ostensibly trusted, can interact with the environment (but is apparently safe). A second, untrusted, cannot interact with the environment. However a third process, although by itself not capable of interaction with the environment, can communicate with both the (supposedly) trusted and the untrusted processes. Now in this case it is conceivably possible for the untrusted process to use the conduit process to corrupt the trusted process. For this reason, we argue that in this situation *all* processes must be considered untrusted because they are all capable of interacting, albeit indirectly (that is, it is transitively closed), with an untrusted source and with the environment. (This example is revisited more formally, with our solution, in example 4.4).

Our solution, based on these assumptions, is to carry an extra set of information through a deduction which we call a *context* (written $\mathbb{C}$). The structure of a context is a mapping of all free names and variables in the process to an annotation representing the trustedness of the process under which that name is (or could be) used (note that this is potentially different to the annotation on the name itself). Then a composition can only be formed between processes where a union can be formed between their respective contexts; for example a trusted process communicating on x would have $x : \mathrm{T}$ in its context, while an

untrusted process might have $x : \mathrm{U}$ in its context and thus the new type rules (section 4) would disallow the two to run in parallel (because the first process can communicate with an untrusted entity so it must also be considered untrusted).

We carry one final bit of information through a deduction; an immutable construct we call the *external context*, written $\mathbb{C}_\epsilon$ (definition 2.13). This represents the names through which communication with the environment can occur (that is, it is a set of names; unlike regular contexts there is no association with trust values). Any composition between processes must consider the external context when calculating the overall trustedness of the two, as must the final deduction (that is, the final rule that calculates the trustedness of a process with respect to the execution environment).

Definition 2.11 (Contexts) $\mathbb{C}$ *is a partial function*

$$\mathbb{C} :: \mathit{datum} \times \mathit{boolexp}$$

Define the total function $\mathbb{C}_T$ *by extension:*

$$\mathbb{C}_T(U) = \begin{cases} b, & \text{if } U : b \in \mathbb{C} \\ \mathrm{T} & \text{otherwise} \end{cases}$$

Definition 2.12 *Write*

$$\mathbb{C}_1 \asymp \mathbb{C}_2 \iff \forall U \in \mathrm{dom}\mathbb{C}_1 \cap \mathrm{dom}\mathbb{C}_2 \Rightarrow \mathbb{C}_1(U) \equiv \mathbb{C}_2(U)$$

The operation $\mathbb{C}_1, \mathbb{C}_2$ *is defined as* $\mathbb{C}_1 \cup \mathbb{C}_2$ *iff* $\mathbb{C}_1 \asymp \mathbb{C}_2$.

Definition 2.13 (External Context)

$$\mathbb{C}_\epsilon :: \mathit{datum}$$

The external context always contains a single "top" element, which is distinct from the set of names and variables contained in processes (and contexts). This is to cover the situation in which there are no *avenues of communication with the environment (and thus the process can be considered trusted); recalling that* $\mathbb{C}(x) = \mathrm{T}$ *if* $x \notin \mathrm{dom}\mathbb{C}$*, so the operation* $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$ *is always defined.*

2.5 Judgements

A valid deduction in the type system (see section 4) is expressed as a judgement; the format of a judgement is shown in definition 2.14:

Definition 2.14 *The judgement*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}$$

says that under the assumption set Γ *the process* P *has the type* [] *(denoting a well-formed process) with trustedness* b*, and has names operating with contexts*

described by the context set $\mathbb{C}$*. The alternate judgement form*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b$$

states as before that under assumptions Γ *process* P *is well-formed and has trustedness* b*; it represents the final outcome of a deduction in which contextual information has been used and is no longer important. Following standard practice,* Γ_x *represents the set* Γ *with all instances of* x *removed.*

The complete set of rules for formulating such a valid deduction are presented in section 4.

3 Adding Certification-Based Run-time Coercion

Now we wish to extend the core calculus just presented with an additional primitive to allow safe coercion: the approach we use is modeled on an example presented by Ørbæk and Palsberg [14], which also serves to illustrate the contrasts with their approach (example 3.1).

Example 3.1 *(From* Trust in the λ-Calculus *[14]) The following code fragment is intended to retrieve a piece of data and a signature from a network, and only use the data if the signature can be verified. The expression "*`trust x`*" means* `x` *is unconditionally trusted:*

```
let (data, sig) = read_from_network in
    if verify_signature(data, sig) then
        handle_event(trust data)
    else
        handle_wrong_signature(data, sig)
```

where the handler code resembles

```
fun handle_event data =
    let trusted_data = check data
    in ...
```

The "`check`*" construct only type-checks on trusted values; which is the case in this example because the signature was verified and the appropriate coercion was made based on the successful result.*

However consider the implications if the programmer accidentally gets the branches in the wrong order; that is:

```
...
    if verify_signature(data, sig) then
        handle_wrong_signature(data, sig)
    else
        handle_event(trust data)
```

According to the compiler this program would still be safe, because due to the presence of the `trust` *cast the* `check` *assertion still succeeds, even though the*

actual verification procedure would have failed.

The approach we take is based on this, but removes the control from the hands of the programmer and places it solely with the verification procedure used. The results of the verification are used to both apply the coercion (to 'trusted' if the integrity can be guaranteed, to 'untrusted' otherwise), and to decide the direction of the branch. In other words, it encapsulates all the elements of the correct code in example 3.1 in a single operation so it cannot possibly be mis-applied.

Our complete syntax incorporating the new construct then becomes (definition 3.2):

Definition 3.2

$$\mathrm{P} ::= \ldots \mid x(U)_{\mathrm{certify}}?\mathrm{P}:\mathrm{Q}$$

3.1 Semantics of certify

First define the following extra judgement form:

Definition 3.3 *Write the following*

$$\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} K : \mathrm{T}$$

to indicate that under the verification scheme named certify *the process or variable K has trustedness* T. *Similarly if K is untrusted under certify.*

We are now in a position to specify the reduction semantics of the certify construct, in the form of an additional two reduction rules:

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} K : \mathrm{T}}{\ldots + \overline{x}[K].\mathrm{R} | x(U)_{\mathrm{certify}}?\mathrm{P}:\mathrm{Q} + \ldots \rightarrow \mathrm{R} | \mathrm{P}\{K/U\}}$$

and the dual

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} K : \mathrm{U}}{\ldots + \overline{x}[K].\mathrm{R} | x(U)_{\mathrm{certify}}?\mathrm{P}:\mathrm{Q} + \ldots \rightarrow \mathrm{R} | \mathrm{Q}\{K/U\}}$$

By way of informal explanation, the new construct behaves like an input process, binding the (single) input to U in P if it is found to be trusted, and to U in Q if found to be untrusted. Note that the details of the certification procedure used are unspecified; by deliberately abstracting in this way it becomes a framework capable of incorporating any method able to determine the integrity of data or code. We note — this will become clearer when we examine the type rules in section 4 — that this removes the possibility for error demonstrated in example 3.1 as the type system will detect any instance of an untrusted variable being used in a trusted environment.

4 The Type System

The complete type rules are shown in figure 1. Many rules contain a term of the form $\mathbb{C}_1, \mathbb{C}_2$ in the consequent. We use this as a form of syntactic shorthand; recalling definition 2.12 this is in fact only defined if $\mathbb{C}_1 \asymp \mathbb{C}_2$: such rules should be read as containing the clause $\mathbb{C}_1 \asymp \mathbb{C}_2$ in the antecedent and either $\mathbb{C}_1, \mathbb{C}_2$ or $\mathbb{C}_1 \cup \mathbb{C}_2$ in the consequent.

$$\frac{}{x : \sigma^b \vdash^{\mathbb{C}_\epsilon} x : \sigma^b; \emptyset}\ (var-1.) \qquad \frac{}{X : []^b \vdash^{\mathbb{C}_\epsilon} X : []^b; X : b}\ (var-2.)$$

$$\frac{}{\emptyset \vdash^{\mathbb{C}_\epsilon} \mathbf{0} : []^b; \emptyset}\ (zero.) \qquad \frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{A} : \sigma^b; \mathbb{C}}{\Gamma, U : \sigma_1^c \vdash^{\mathbb{C}_\epsilon} \mathrm{A} : \sigma^b; \mathbb{C}}\ (weak.)$$

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}}{\Gamma \vdash^{\mathbb{C}_\epsilon} !\mathrm{P} : []^b; \mathbb{C}}\ (repl.) \qquad \frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}_1 \quad \Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^c; \mathbb{C}_2}{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P}|\mathrm{Q} : []^{\bigwedge_{x \in \mathbb{C}_\epsilon} (\mathbb{C}_1, \mathbb{C}_2)_T(x)}; \mathbb{C}_1, \mathbb{C}_2}\ (comp.)$$

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}}{\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\nu U)\mathrm{P} : []^b; \mathbb{C}_U}\ (res.) \qquad \frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}_1 \quad \Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_2}{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} + \mathrm{Q} : []^b; \mathbb{C}_1, \mathbb{C}_2}\ (sum.)$$

$$\frac{\Gamma, x : (b \cdot \vec{\sigma^c})^b, \vec{U} : b \cdot \vec{\sigma^c} \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^d; \mathbb{C} \quad \forall U \in \vec{U}. d = \mathbb{C}(U)}{\Gamma_{\vec{U}}, x : (b \cdot \vec{\sigma^c})^b \vdash^{\mathbb{C}_\epsilon} x(\vec{U}).\mathrm{P} : []^d; \mathbb{C}_{\vec{U}}, x : d}\ (inp.)$$

$$\frac{\Gamma, x : (bc \cdot \vec{\sigma^d})^b \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^c; \mathbb{C}_1 \quad \Gamma, x : (bc \cdot \vec{\sigma^d})^b \vdash^{\mathbb{C}_\epsilon} \vec{K} : bc \cdot \vec{\sigma^d}; \mathbb{C}_2}{\Gamma_x, x : (bc \cdot \vec{\sigma^d})^b \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{P} : []^c; \mathbb{C}_1, x : c, \mathbb{C}_2, FV\left(\vec{K}\right) : c}\ (out.)$$

$$\frac{\begin{array}{c}\Gamma, x : (\sigma^b)^d, U : \sigma^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^c; \mathbb{C}_1 \quad \mathbb{C}_1(U) = c \\ \Gamma, x : (\sigma^b)^d, U : \sigma^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^c; \mathbb{C}_2 \quad \mathbb{C}_2(U) = c\end{array}}{\Gamma_U, x : (\sigma^b)^d \vdash^{\mathbb{C}_\epsilon} x(U)_{\text{certify}}^{?}\mathrm{P}{:}\mathrm{Q} : []^c; \mathbb{C}_{1U}, \mathbb{C}_{2U}, x : c}\ (cert.)$$

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}}{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^{\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)}}\ (final)$$

Fig. 1. Type Rules

Some explanation follows: several of the rules are fairly standard and relatively unchanged from their un-annotated versions [20]. A null process can be given any trust value for the purposes of a deduction, however because it does nothing it has an empty context (note that because of this in a complete deduction — one ending with the final rule — even an untrusted null process would still be trusted overall; a logical outcome due to its inactivity. See example 4.2). We require separate variable axioms; because a variable is a process it has a context (of its own annotation) associated with it, while a name does not. Weakening is allowed, although note that no additional information is added to the context which is entirely dependent on the term.

Replication does not change the typing; while restriction, being a binding operation, removes the bound name or variable from both the assumption set and the context. Summation (choice) is permitted so long as the types match and a union can be formed between their respective contexts. The requirement on the types (annotations) matching is due to the intuition that in a choice construct we do not know which will reduce (discarding the alternatives), so it is desirable that no matter which is chosen it has the same type.

Other rules require closer attention. The overall trustedness in a composition construction is determined by the possibility for interaction with the external environment of both processes (the construct $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$); as is the final rule, which also drops the context as it is no longer required (since it is always the final rule in a deduction).

The input rule requires that, to enforce policies on arities and types carried, the types of names being input match the types carried by the input channel; with the additional requirement that the types carried must be multiplied by the trustedness of the channel. That is, since T is the multiplicative identity a trusted channel can carry values of all trustedness, but because $b \cdot \mathrm{U} = \mathrm{U}$ all data carried by an untrusted channel must be considered untrusted; matching our intuition. There is an additional requirement on the input; that all names being bound appear in a context of the same trustedness as the overall trustedness of the process. This prevents the situation in which a name in an untrusted context, in an otherwise-trusted process, is bound to a name that can communicate with the environment and thus corrupts the process (it might appear that a context *at least as safe* would suffice and be more flexible, that is $\mathbb{C}(U) \geq c$; however this breaks our subject reduction property, presented in section 6.2). The output rule similarly requires that the types of the output data match those carried by the channel, with a similar requirement that their trustedness be multiplied by the trustedness of the channel. Unlike the input case however, there is an additional multiplication by the trustedness of the sending *process*; this ensures that all data sent by an untrusted process is itself untrusted, even if transmitted on a trusted channel.

The certify rule takes two processes, one constructed using a trusted variable, the other an identically-named untrusted variable. If the input data is found to be trusted it is bound to a trusted variable in P, otherwise to an untrusted variable in Q. Because it is a binding (on U) construct, information concerning U is removed from the contexts and assumption sets. There is also a requirement similar to the input rule that the name/variable being bound is in a context the same as the overall trustedness.

Example 4.1 *In example 3.1 we presented an example from a different system and outlined our reservations; notably that a simple programmer error could have dangerous safety repercussions with no warning from the compiler. Here we briefly illustrate how equivalent code might look in our system, and how the type system prevents a similar form of error from occurring. If we assume the data to be tested is received along a channel x (replacing*

`read_from_network`*), then for some overall trustedness* b *we have*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} x(U)_{\text{certify}}?\mathit{handleData}{:}\mathit{Err} : []^b; \mathbb{C}$$

We note first of all that through the modularity of the certify framework we do not explicitly mention the use of a signature to determine trustedness; it may just as well be determined through use of code analysis for example. The second, and most important benefit of our system is apparent when we consider the antecedents to the above deduction:

$$\Gamma, x : (\sigma^c)^d, U : \sigma^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathit{handleData} : []^b; \mathbb{C}_1$$
$$\Gamma, x : (\sigma^c)^d, U : \sigma^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathit{Err} : []^b; \mathbb{C}_2$$

(where $\mathbb{C} = x : b, \mathbb{C}_1, \mathbb{C}_2$*) In other words, the certify deduction* cannot *be formed if the branches are confused, as the bound variable must be trusted in the first branch, and untrusted in the second.*

The final rule in any complete deduction is the final rule; similarly to the composition rule, it combines the information collected in the contexts with the environmental information contained in the external context to compute the overall trustedness of a process. Because it is the final rule in any deduction, the contextual information can be discarded.

Example 4.2 *Consider the deduction* $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathbf{0} : []^{\mathrm{U}}; \emptyset$*, which holds for any* Γ *(through repeated application of the variable rules). A null process can be given any trustedness value for the purposes of a deduction; however under the final rule it can have only one value:*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \mathbf{0} : []^{\mathrm{T}}$$

since $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) = \mathrm{T}$ *for all* $\mathbb{C}_\epsilon$ *when* $\mathbb{C} = \emptyset$*; a logical result since a null process — by virtue of being inactive — cannot corrupt the environment.*

Another example, illustrating a malicious but impotent agent in the network:

Example 4.3 *Consider a trusted agent* P *(and assume for simplicity that* P *has no free names); then it is easy to verify that the following deduction is valid:* $\Gamma \vdash^{\mathbb{C}_\epsilon} x.\mathrm{P} : []^{\mathrm{T}}; x : \mathrm{T}$*. We can similarly derive an untrusted (and also closed) agent* $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{\mathrm{U}}; \mathbb{C}$*, and hence* $\Gamma \vdash^{\mathbb{C}_\epsilon} y.\mathrm{Q} : []^{\mathrm{U}}; y : \mathrm{U}$*. If the two co-exist in the same environment, and the only port of communication with the environment is* x *(that is,* $\mathbb{C}_\epsilon = x$*) the following is easy to verify:*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} x.\mathrm{P} | y.\mathrm{Q} : []^{\mathrm{T}}$$

A third, slightly similar but more complicated example showing how a dangerous agent is detected by the type system:

Example 4.4 *We again assume a trusted and closed agent* P*, but this time with a slightly more complicated sequence of (possible) communications added:*

$\Gamma \vdash^{\mathbb{C}_\epsilon} x(v).\overline{z}[v].\mathrm{P} : []^{\mathrm{T}}; x : \mathrm{T}, z : \mathrm{T}$. *Again, assume an untrusted agent with the following typing:* $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{y}[w].\mathrm{Q} : []^{\mathrm{U}}; y : \mathrm{U}, w : \mathrm{U}$. *With the port of observation* ($\mathbb{C}_\epsilon$) *again* x*, it is easy to verify that no possibility for corruption exists:*

$$\Gamma \vdash^{\mathbb{C}_\epsilon} x(v).\overline{z}[v].\mathrm{P} \mid \overline{y}[w].\mathrm{Q} : []^{\mathrm{T}}$$

However, if a third agent is introduced into the system the situation changes: given $\Gamma \vdash^{\mathbb{C}_\epsilon} z(v).y(w).\mathrm{R} : []^b; z : b, y : b$ *there is now no valid typing that will allow the three to run in parallel. To confirm this, recall from the composition rule that we must be able to form the union of the individual contexts; there is no* b *such that* $\{x : \mathrm{T}, z : \mathrm{T}, v : \mathrm{T}\} \asymp \{z : b, y : b\} \asymp \{y : \mathrm{U}, w : \mathrm{U}\}$ *is defined since we must have both* $b = \mathrm{T}$ *and* $b = \mathrm{U}$*. This confirms our motivation (described in section 2.4) that because the untrusted process now has a potential avenue of communication — albeit indirect — with the external environment the entire group must be considered untrusted.*

Example 4.5 *As one final example, we examine the subtleties involved in a higher-order example; sending a process. First of all note that the trustedness of a name and its contextual trustedness are not necessarily the same; a name may itself be untrusted but only appear in a trusted context. The following scenario, that of a remote host sending an agent to be executed by a local host, provides a realistic yet easy to follow framework for analysis:*

$$x(X).\,(host \mid X) \mid \overline{x}[agent].remote$$

We note a few possible (and rejected) typings for such a situation. First of all, both parties must have the same trustedness in order to be able to communicate; this is as a consequence of having to form the union of the respective contexts, which can only occur if x *has the same contextual value. Secondly, even if both agents are trusted, the channel of communication between them can itself be untrusted; all data transmitted would also be untrusted. Closer examination of this point however reveals that although trusted processes can send untrusted names, they cannot in most (useful) cases send untrusted* agents*. To see why this is the case consider the construction of the output rule, and in particular the requirements on the contexts. Because the argument must be multiplied by the trustedness of both the sending process and the channel, "agent" must be untrusted if* x *is too. This means that in the deduction* $\Gamma \vdash^{\mathbb{C}_\epsilon} agent : []^{\mathrm{U}}; \mathbb{C}_a$ *most names in the domain of* $\mathbb{C}_a$ *will have a value of* U*; however the output role requires that* all *values in the range of* $\mathbb{C}_a$ *have the same trustedness as that of "remote" (in this case* T*) so such a process cannot be typed under those conditions. This is perhaps a little restrictive; however it is certainly the safest option as it prevents a potentially corrupted process from becoming a conduit as it in example 4.4. We suspect that the introduction of subtyping (see section 7.2 for possibilities) might allow a little more flexibility. In summary though, when processes are sending agents (names are not affected) the trustedness of the channel must be the same as that of the sending process.*

5 Implementation

5.1 Unification

Numerous unification algorithms will be needed to handle the different combinations of base types and annotations present, although they all share a common structure. We begin by defining substitutions:

Definition 5.1

- *A substitution is a pair* $(\mathbb{S}_T : Type \to Type, \mathbb{R}_B : boolexp \to boolexp)$.
- *Usually written just as* $\mathbb{S}$*; sometimes write* $\mathbb{R}$ *to denote* $(\mathrm{Id}, \mathbb{R}_B)$.
- *Write* Id *for* $(\mathrm{Id}, \mathrm{Id})$.
- *Write* $\mathbb{S}_1;\mathbb{S}_2$ *for* $\mathbb{S}_2 \circ \mathbb{S}_1$ *(that is;* $\mathbb{S}_1;\mathbb{S}_2(\sigma^b) \equiv \mathbb{S}_2(\mathbb{S}_1(\sigma^b))$*)*
- *Application: if* $\mathbb{S} = (\mathbb{S}_T, \mathbb{R}_B)$ *then*
  · $\mathbb{S}(\sigma^b) = \mathbb{S}_T;\mathbb{R}_B(\sigma^b)$
  · $\mathbb{S}((\sigma_1^b \ldots \sigma_n^c)^d) = (\mathbb{S}(\sigma_1^b) \ldots \mathbb{S}(\sigma_n^c))^{\mathbb{S}(d)}$
- *Given* $\mathbb{S} = (\mathbb{S}_T, \mathbb{R}_B)$*, write* $\mathbb{S}[\alpha^i := \mathrm{S}^j]$ *for* $(\mathbb{S}_T[\alpha^i := \mathrm{S}^j], \mathbb{R}_B)$ *and* $\mathbb{S}[i := j]$ *for* $(\mathbb{S}_T, \mathbb{R}_B[i := j])$*. Similarly, write* $\mathbb{S};\mathbb{R}$ *for* $(\mathbb{S}_T, \mathbb{R}_B;\mathbb{R})$.

The following two algorithms are the most important, unifying boolean annotations and types (which include boolean annotations) respectively. The remainder exploit the first two (5.2 and 5.3) to work over sets.

Definition 5.2 BUNIFY *is the boolean unifier [7] that returns the most general unifier of its arguments if one exists, or it fails.*

Definition 5.3 $\mathcal{U}$ *is the most general unifier of a pair of types, see for example Huet [6]. It uses* BUNIFY *to unify the annotations:*

$$
\begin{aligned}
\mathcal{U}\big(\alpha^b, \sigma^c\big) &= \textit{if } \alpha \in \sigma \wedge \alpha \neq \sigma \textit{ then } \bot \textit{ else} \\
&\qquad \textit{let } \mathbb{S} = \mathrm{Id}[\alpha := \sigma] \textit{ in } \mathbb{S};\mathrm{BUNIFY}(b, c) \\
\mathcal{U}\big(\sigma^c, \alpha^b\big) &= \mathcal{U}\big(\alpha^b, \sigma^c\big) \\
\mathcal{U}\big((\vec{\sigma^c})^b, (\vec{\sigma^e})^d\big) &= \textit{if } \mathit{length}\big(\vec{\sigma^c}\big) \neq \mathit{length}\big(\vec{\sigma^e}\big) \textit{ then } \bot \textit{ else} \\
&\qquad \textit{let } \mathbb{R} = \mathrm{BUNIFY}(b, d) \textit{ in } \mathbb{R};\mathcal{U}\big(\mathbb{R}(\vec{\sigma^c}), \mathbb{R}(\vec{\sigma^e})\big) \\
\mathcal{U}\big(\sigma^b\vec{\sigma^c}, \sigma^d\vec{\sigma^e}\big) &= \textit{if } \mathit{length}\big(\vec{\sigma^c}\big) \neq \mathit{length}\big(\vec{\sigma^e}\big) \textit{ then } \bot \textit{ else} \\
&\qquad \textit{let } \mathbb{S} = \mathcal{U}\big(\sigma^b, \sigma^d\big) \textit{ in } \mathbb{S};\mathcal{U}\big(\mathbb{S}(\vec{\sigma^c}), \mathbb{S}(\vec{\sigma^e})\big) \\
\mathcal{U}\big([]^b, []^c\big) &= \mathrm{BUNIFY}(b, c)
\end{aligned}
$$

Definition 5.4 Unify *unifies two assumption sets, by unifying the types of all names present in both arguments:*

$$\mathrm{Unify}(\Gamma_1, \Gamma_2) \triangleq \mathrm{Id} \quad (\mathrm{dom}\Gamma_1 \cap \mathrm{dom}\Gamma_2 = \emptyset)$$
$$\mathrm{Unify}\big(\{x : \sigma_1^b\} \cup \Gamma_1, \{x : \sigma_2^c\} \cup \Gamma_2\big) \triangleq \mathbb{S}; \mathrm{Unify}(\mathbb{S}(\Gamma_1), \mathbb{S}(\Gamma_2)) \quad \textit{where } \mathbb{S} = \mathcal{U}\big(\sigma_1^b, \sigma_2^c\big)$$

Definition 5.5 CUnify *unifies two contexts, by unifying the mappings of all names present in both arguments:*

$$\mathrm{CUnify}(\mathbb{C}_1, \mathbb{C}_2) \triangleq \mathrm{Id} \quad (\mathrm{dom}\mathbb{C}_1 \cap \mathrm{dom}\mathbb{C}_2 = \emptyset)$$
$$\mathrm{CUnify}(\{x : b\} \cup \mathbb{C}_1, \{x : c\} \cup \mathbb{C}_2) \triangleq \mathbb{R}; \mathrm{CUnify}(\mathbb{R}(\mathbb{C}_1), \mathbb{R}(\mathbb{C}_2)) \quad \textit{where } \mathbb{R} = \mathrm{BUNIFY}(b, c)$$

Finally, enforcing the multiplication constraint required by some type rules is implemented by the following algorithm:

Definition 5.6

$$\mathbb{M}\big(b, \sigma^{\mathrm{T}}\big) = \mathrm{BUNIFY}(b, \mathrm{T})$$
$$\mathbb{M}\big(b, \sigma^{\mathrm{U}}\big) = \mathrm{Id}$$
$$\mathbb{M}(b, \sigma^c) = \mathrm{BUNIFY}(c, b \cdot i)$$
$$\mathbb{M}\Big(b, \sigma^c \vec{\sigma^{!e}}\Big) = \textit{let } \mathbb{S} = \mathbb{M}(b, \sigma^c) \textit{ in } \mathbb{S}; \mathbb{M}\Big(\mathbb{S}(b), \mathbb{S}(\vec{\sigma^{!e}})\Big)$$

5.2 The Type Inference Algorithm

The complete type inference algorithm is split over figures 2 and 3. The auxiliary algorithm of figure 4 is used to unify lists of processes or names, such as those that may be found as the "arguments" of an output.

6 Safety

There are two main components we wish to prove in demonstrating the safety and correctness of our system; that subject reduction holds for the type system, and that the algorithm is sound and correct.

6.1 Properties of Contexts

First we need the following properties of contexts (proofs are all by induction):

Lemma 6.1 (Context Subset)

$$\forall \mathbb{C}' \subseteq \mathbb{C}. \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$$

Proof. Let $\vec{y}$ be the sequence of names given by $\mathbb{C}_\epsilon \cap \mathrm{dom}\mathbb{C}$. Then since $\mathbb{C}' \subseteq \mathbb{C}$ the sequence $\vec{y'} = \mathbb{C}_\epsilon \cap \mathrm{dom}\mathbb{C}'$ is a subsequence of $\vec{y}$, and hence by

$$
\begin{aligned}
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathbf{0}) &= \left\langle \emptyset, \emptyset, []^i \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(x) &= \left\langle \{x : \alpha^i\}, \emptyset, \alpha^i \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(X) &= \left\langle \{X : []^i\}, \{X : i\}, []^i \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(!\mathrm{P}) &= \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}((\nu U)\mathrm{P}) &= \text{let } \left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
&\quad \text{in } \left\langle \Gamma_U, \mathbb{C}_U, []^b \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P} + \mathrm{Q}) &= \text{let } \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
&\qquad \langle \Gamma_2, \mathbb{C}_2, []^c \rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{Q}) \\
&\qquad \mathbb{R} = \mathrm{BUNIFY}(b, c) \\
&\qquad \mathbb{S}_1 = \mathbb{R}; \mathrm{Unify}(\mathbb{R}(\Gamma_1), \mathbb{R}(\Gamma_2)) \\
&\qquad \mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2)) \\
&\quad \text{in } \left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2), []^{\mathbb{S}_2(b)} \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}|\mathrm{Q}) &= \text{let } \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
&\qquad \langle \Gamma_2, \mathbb{C}_2, []^c \rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{Q}) \\
&\qquad \mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2) \\
&\qquad \mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2)) \\
&\qquad \mathbb{C} = \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2) \\
&\quad \text{in } \left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{C}, []^{\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)} \right\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}\Big(x(\vec{U}).\mathrm{P}\Big) &= \text{let } \left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
&\qquad \mathbb{S}_1 = \mathrm{Unify}\Big(\Gamma, \{x : (\vec{\alpha^i})^j, \vec{U} : \vec{\alpha^i}\}\Big) \\
&\qquad \mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\Big(\mathbb{S}_1(j), \mathbb{S}_1((\vec{\alpha^i})^j)\Big) \\
&\qquad \mathbb{S}_3 = \mathbb{S}_2; \mathrm{CUnify}(\mathbb{S}_2(\mathbb{C}), \mathbb{S}_2(\{x : b\})) \\
&\quad \text{in } \left\langle \mathbb{S}_3(\Gamma_{\vec{U}} \cup \{x : (\vec{\alpha^i})^j\}), \mathbb{S}_3(\mathbb{C}_{\vec{U}} \cup \{x : b\}), []^{\mathbb{S}_3(b)} \right\rangle
\end{aligned}
$$

Fig. 2. Type Inference Algorithm

definition $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$. (since $\forall b, c. b \cdot c \leq c$, and where $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$ and $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$ are the sequences $\bigwedge \vec{b}$ and $\bigwedge \vec{b'}$, corresponding to the mappings of $\vec{y}$ and $\vec{y'}$ in $\mathbb{C}$ and $\mathbb{C}'$ respectively).

Lemma 6.2 (Context Expansion) *Let $\mathbb{C}' = \mathbb{C}, y : b$ where $\mathbb{C}$ is any context such that $y \notin \mathrm{dom}\mathbb{C}$, then*

$$\forall \mathbb{C}, b, y \notin \mathrm{dom}\mathbb{C}. b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$$

$$
\begin{aligned}
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}\left(\overline{x}[\vec{K}].\mathrm{P}\right) = \text{let } & \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
& \left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma^c} \right\rangle = *\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}\left(\vec{K}\right) \\
& \mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2) \\
& \mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\big(\mathbb{S}_1(b), \mathbb{S}_1(\vec{\sigma^c})\big) \\
& \mathbb{S}_3 = \mathbb{S}_2; \mathbb{M}\big(\mathbb{S}_2(i), \mathbb{S}_2((\vec{\sigma^c})^i)\big) \\
& \mathbb{S}_4 = \mathbb{S}_3; \mathrm{Unify}\big(\mathbb{S}_3(\Gamma_1 \cup \Gamma_2), \mathbb{S}_3(\{x : (\vec{\sigma^c})^i\})\big) \\
& \mathbb{S}_5 = \mathbb{S}_4; \mathrm{CUnify}\Big(\mathbb{S}_4(\mathbb{C}_2), \mathbb{S}_4(FV\left(\vec{K}\right) : b, x : b)\Big) \\
& \mathbb{S}_6 = \mathbb{S}_5; \mathrm{CUnify}\Big(\mathbb{S}_5(\mathbb{C}_1), \mathbb{S}_5(FV\left(\vec{K}\right) : b, x : b)\Big) \\
\text{in } & \Big\langle \mathbb{S}_6(\Gamma_1 \cup \Gamma_2 \cup \{x : (\vec{\sigma^c})^i), \\
& \qquad \mathbb{S}_6(\mathbb{C}_1 \cup \{FV\left(\vec{K}\right) : b\} \cup \{x : b\}), []^{\mathbb{S}_6(b)} \Big\rangle \\
\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(x(U)_{\mathrm{certify}}?\mathrm{P}{:}\mathrm{Q}) = \text{let } & \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{P}) \\
& \langle \Gamma_2, \mathbb{C}_2, []^c \rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(\mathrm{Q}) \\
& \mathbb{S}_1 = \mathrm{Unify}\big(\Gamma_1, \{U : \alpha^{\mathrm{T}}\}\big) \\
& \mathbb{S}_2 = \mathbb{S}_1; \mathrm{Unify}\big(\mathbb{S}_1(\Gamma_2), \{U : \mathbb{S}_1(\alpha^{\mathrm{U}})\}\big) \\
& \mathbb{S}_3 = \mathbb{S}_2; \mathrm{Unify}(\mathbb{S}_2(\Gamma_{1U}), \mathbb{S}_2(\Gamma_{2U})) \\
& \mathbb{S}_4 = \mathbb{S}_3; \mathrm{Unify}\big(\mathbb{S}_3(\Gamma_{1U} \cup \Gamma_{2U}), \mathbb{S}_3(\{x : (\alpha^i)^j\})\big) \\
& \mathbb{S}_5 = \mathbb{S}_4; \mathrm{CUnify}(\mathbb{S}_4(\mathbb{C}_{1U}), \mathbb{S}_4(\mathbb{C}_{2U})) \\
& \mathbb{S}_6 = \mathbb{S}_5; \mathrm{CUnify}(\mathbb{S}_5(\mathbb{C}_{1U}, \mathbb{C}_{2U}), \mathbb{S}_5(x : b)) \\
& \mathbb{S}_7 = \mathbb{S}_6; \mathrm{BUNIFY}(\mathbb{S}_6(b), \mathbb{S}_6(c)) \\
\text{in } & \big\langle \mathbb{S}_7(\Gamma_{1U} \cup \Gamma_{2U} \cup \{x : (\alpha^i)^j\}), \\
& \qquad \mathbb{S}_7(\mathbb{C}_{1U} \cup \mathbb{C}_{2U} \cup \{x : b\}), []^{\mathbb{S}_7(b)} \big\rangle
\end{aligned}
$$

Fig. 3. Type Inference Algorithm (continued)

$$
\begin{aligned}
*\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(K) &= \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(K) \\
*\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}\left(K\vec{K'}\right) = \text{let } & \left\langle \Gamma_1, \mathbb{C}_1, \sigma_1^b \right\rangle = \mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}(K) \\
& \left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma_2^c} \right\rangle = *\mathrm{Type}_{\pi}^{\mathbb{C}_\epsilon}\left(\vec{K'}\right) \\
& \mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2) \\
& \mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2)) \\
\text{in } & \left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2), \mathbb{S}_2(\sigma_1^b \vec{\sigma_2^c}) \right\rangle
\end{aligned}
$$

Fig. 4. Type Inference Algorithm for Multiple Processes

Proof. Let $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) = c$ (and from the statement, $b \leq c$). Then given $\mathbb{C}' = \mathbb{C}, y : b$ ($y \notin \text{dom}\mathbb{C}$) we have

$$\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x) = \begin{cases} c, & \text{if } y \notin \mathbb{C}_\epsilon \\ b \cdot c, & \text{if } y \in \mathbb{C}_\epsilon \end{cases}$$

and since $b \leq c \Rightarrow b = b \cdot c$ we have $b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$ as required.

The next corollary follows straight-forwardly from the previous result:

Corollary 6.3

$$\exists b. \forall \mathbb{C}, y \notin \text{dom}\mathbb{C}, c \geq b. b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \Rightarrow b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$$

where $\mathbb{C}' = \mathbb{C}, y : c$.

Lemma 6.4 (Context Join)

$$b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{1T}(x), b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{2T}(x) \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} (\mathbb{C}_1, \mathbb{C}_2)_T(x)$$

Proof. By definition of the comma operator and straightforward induction on the contents of $\mathbb{C}_1$, $\mathbb{C}_2$.

The following is a direct result of the output type rule:

Fact 6.5 (Output Context)

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{P} : []^b; \mathbb{C} \Rightarrow \forall y \in FV\left(\vec{K}\right). \mathbb{C}(y) = b$$

Lemma 6.6 (Context Annotation)

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C} \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$$

Proof. By induction on the derivation of $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}$.

- $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}$: Trivial.
- $\Gamma \vdash^{\mathbb{C}_\epsilon} !\mathrm{P} : []^b; \mathbb{C}$: By the induction hypothesis.
- $\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\nu U)\mathrm{Q} : []^b; \mathbb{C}$: By the induction hypothesis the lemma holds for Q; i.e. $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C} \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. They by the restriction type rule we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_U$. Then we have $\mathbb{C}_U \subseteq \mathbb{C}$ and so by the context subset lemma (6.1) $\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{UT}(x)$ and hence $b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{UT}(x)$.
- $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}|\mathrm{R} : []^b; \mathbb{C}$: By definition; the composition type rule states that $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}|\mathrm{R} : []^b; \mathbb{C}$ where $b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$.

- $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} + \mathrm{R} : []^b; \mathbb{C}$: By the induction hypothesis the lemma holds for Q and R individually; i.e. $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_1 \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{1U}(x)$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^b; \mathbb{C}_2 \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{2U}(x)$. Hence by the summation type rule and the context join lemma (6.4) $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} + \mathrm{R} : []^b; \mathbb{C}_1, \mathbb{C}_2 \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} (\mathbb{C}_1, \mathbb{C}_2)_T(x)$.
- $\Gamma \vdash^{\mathbb{C}_\epsilon} x(\vec{U}).\mathrm{Q} : []^b; \mathbb{C}$: By the induction hypothesis and the input type rule $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C} \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. Then by the input type rule we have $\Gamma_{\vec{U}} \vdash^{\mathbb{C}_\epsilon} x(\vec{U}).\mathrm{Q} : []^b; \mathbb{C}_{\vec{U}}, x : b$ and so by the context subset lemma (6.1) and the context expansion lemma (6.2) $b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} (\mathbb{C}_{\vec{U}}, x : b)_T(x)$.
- $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{Q} : []^b; \mathbb{C}$: By the induction hypothesis and the output type rule $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C} \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. Then by the output type rule we get $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{Q} : []^b; \mathbb{C}, FV\left(\vec{K}\right) : b, x : b$ and hence by the context expansion lemma (6.2) and output context fact (6.5) $b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$ where $\mathbb{C}' = \mathbb{C}, FV\left(\vec{K}\right) : b, x : b$.
- $\Gamma \vdash^{\mathbb{C}_\epsilon} x(U)_{\mathrm{certify}}?\mathrm{Q}{:}\mathrm{R} : []^b; \mathbb{C}$: By the induction hypothesis, the lemma holds individually:

$$\Gamma, x : (\sigma^c)^d, U : \sigma^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_1 \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{1T}(x)$$

$$\Gamma, x : (\sigma^c)^d, U : \sigma^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^b; \mathbb{C}_2 \Rightarrow b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}_{2T}(x)$$

  then by the certify type rule $\Gamma \vdash^{\mathbb{C}_\epsilon} x(U)_{\mathrm{certify}}?\mathrm{Q}{:}\mathrm{R} : []^b; \mathbb{C}_1, \mathbb{C}_2, x : b$; and so by the context expansion lemma (6.2) and the context join lemma (6.4) $b \leq \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$ where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2, x : b$.

Finally, we will need a boolean substitution defined on the reduction path:

Definition 6.7 *See figure 5 for a full definition; we sketch the general result here. Given the following:*

$$\mathrm{P} \rightarrow \mathrm{P}' \Vdash^{\Gamma}_{\mathrm{certify}} \mathbb{R}$$

Then $\mathbb{R}$ *is a boolean substitution representing exactly every coercion involved in the reduction path. It is defined by induction on the reduction path, and would be the identity substitution in a reduction not involving any certify constructs.*

6.2 Subject Reduction

Now we are in a position to formulate our subject reduction statement, starting with a substitution lemma:

$$\frac{}{\mathrm{P} \to \mathrm{P} \Vdash^{\Gamma}_{\text{certify}} \mathrm{Id}}\ (\textit{reflex.}) \qquad \frac{}{\overline{x}[\vec{K}].\mathrm{P} \xrightarrow{\overline{x}[\vec{K}]} \mathrm{P} \Vdash^{\Gamma}_{\text{certify}} \mathrm{Id}}\ (\textit{out.})$$

$$\frac{\mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}}{\mathrm{P} + \mathrm{Q} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}}\ (\textit{sum.}) \qquad \frac{}{x(\vec{U}).\mathrm{P} \xrightarrow{x(\vec{K})} \mathrm{P}\{\vec{K}/\vec{U}\} \Vdash^{\Gamma}_{\text{certify}} \mathrm{Id}}\ (\textit{inp.})$$

$$\frac{\mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_1 \quad \mathrm{Q} \to \mathrm{Q}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_2}{\mathrm{P}|\mathrm{Q} \to \mathrm{P}'|\mathrm{Q}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_1; \mathbb{R}_2}\ (\textit{comp.})$$

$$\frac{\mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_1 \quad \mathrm{P}' \to \mathrm{P}'' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_2}{\mathrm{P} \to \mathrm{P}'' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_1; \mathbb{R}_2}\ (\textit{trans.})$$

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} K : \sigma^i; \mathbb{C} \quad \Gamma \vdash^{\mathbb{C}_\epsilon}_{\textit{certify}} K : \mathrm{T}}{x(U)_{\text{certify}}\mathbf{?}\mathrm{P}{:}\mathrm{Q} \xrightarrow{x(K)} \mathrm{P}\{K/U\} \Vdash^{\Gamma}_{\text{certify}} \mathrm{Id}[i := \mathrm{T}]}\ (\textit{cert} - \mathrm{T})$$

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} K : \sigma^i; \mathbb{C} \quad \Gamma \vdash^{\mathbb{C}_\epsilon}_{\textit{certify}} K : \mathrm{U}}{x(U)_{\text{certify}}\mathbf{?}\mathrm{P}{:}\mathrm{Q} \xrightarrow{x(K)} \mathrm{Q}\{K/U\} \Vdash^{\Gamma}_{\text{certify}} \mathrm{Id}[i := \mathrm{U}]}\ (\textit{cert} - \mathrm{U})$$

Fig. 5. Reduction Substitution Relation

Lemma 6.8 (Substitution Lemma) $\exists \mathbb{C}'_2 \supseteq \mathbb{C}_2$.

$$\frac{\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : \sigma^b_1; \mathbb{C}_1 \qquad \Gamma_U \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_2 \quad \forall V \in FV(K).(\mathbb{C}_1, \mathbb{C}_2)(V) = \mathbb{C}_1(U) \qquad \mathbb{C}_1(U) = b}{\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{P}\{K/U\} : \sigma^b_1; \mathbb{C}_{1U}, \mathbb{C}'_2}$$

We make two extra requirements than the usual statement, to cater for the extra information provided by the contexts. As with the input rule we require that the variable being substituted appears in a context at least as safe as the overall trustedness, and we also require that every free variable in the new name (process) appears in a context identical to the substitution variable. The proof of subject reduction — which requires the substitution lemma — reveals that every substitution which occurs due to a reduction does indeed appear in such a situation.

Proof. By induction on the structure of P and the in certain cases on the type of U and K.

- $\mathrm{P} \equiv \mathbf{0}$: Trivial.
- $\mathrm{P} \equiv X$: Trivial; by definition. Note that by the statement, $\Gamma_X, X : []^b \vdash^{\mathbb{C}_\epsilon} X : []^b; \mathbb{C}_1$ where $\mathbb{C}_1 = X : b$, and $\Gamma_X \vdash^{\mathbb{C}_\epsilon} K : []^b; \mathbb{C}_2$ where $\forall V \in FV(K).\mathbb{C}_2(V) = \mathbb{C}_1(X) = b$ (and $b \geq b$ by definition). Then by

the definition of substitution we have $\Gamma_X \vdash^{\mathbb{C}_\epsilon} K : []^b; \mathbb{C}_{1X}, \mathbb{C}'_2$ as required (where $\mathbb{C}_{1X} = \emptyset$ and $\mathbb{C}'_2 = \mathbb{C}_2$).

- $\mathrm{P} \equiv !\mathrm{P}'$: By definition $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} !\mathrm{P}' : []^b; \mathbb{C}_1$ and hence by the replication type rule $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_1$. By the induction hypothesis the lemma holds for P', i.e. given $\Gamma_U \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_2$ where $\forall V \in FV(K).(\mathbb{C}_1, \mathbb{C}_2)(V) = \mathbb{C}_1(U)$ and $\mathbb{C}_1(U) \geq b$, then $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^b; \mathbb{C}_{1U}, \mathbb{C}'_2$ (with $\mathbb{C}_2 \subseteq \mathbb{C}'_2$). Hence by the replication type rule we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} !\mathrm{P}'\{K/U\} : []^b; \mathbb{C}_{1U}, \mathbb{C}'_2$ as required.
- $\mathrm{P} \equiv (\nu V)\mathrm{P}'$: By definition $\Gamma_{UV}, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} (\nu V)\mathrm{P}' : []^b; \mathbb{C}_{1V}$ and hence by the restriction type rule $\Gamma_U, U : \sigma^c, V : \sigma^d_1 \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_1$. By the induction hypothesis the lemma holds for P', i.e. given $\Gamma_U \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_2$ (note that since we are assuming capture-avoiding substitutions, i.e. that all bound names are renamed to be distinct, we don't need to consider the restricted variable V in the deduction of K) where $\forall V \in FV(K).(\mathbb{C}_1, \mathbb{C}_2)(V) = \mathbb{C}_1(U)$ and $\mathbb{C}_1(U) \geq b$, then $\Gamma_U, V : \sigma^d_1 \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^b; \mathbb{C}_{1U}, \mathbb{C}'_2$ (with $\mathbb{C}_2 \subseteq \mathbb{C}'_2$). Hence by the restriction type rule we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\nu V)\mathrm{P}'\{K/U\} : []^b; \mathbb{C}_{1UV}, \mathbb{C}'_{2V}$ as required.
- $\mathrm{P} \equiv \mathrm{P}' + \mathrm{Q}$: By definition, $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' + \mathrm{Q} : []^b; \mathbb{C}$ and hence by the summation type rule, $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_1$ and $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_2$ where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$. By the induction hypothesis, the lemma holds for both P' and Q, i.e. given $\Gamma_U \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_3$ we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^b; \mathbb{C}_{1U}, \mathbb{C}'_3$ and $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}\{K/U\} : []^b; \mathbb{C}_{2U}, \mathbb{C}''_3$. Then by the summation type rule and the definition of substitution we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\mathrm{P}' + \mathrm{Q})\{K/U\} : []^b; \mathbb{C}_{1U}, \mathbb{C}_{2U}, \mathbb{C}'_3, \mathbb{C}''_3$ as required.
- $\mathrm{P} \equiv \mathrm{P}'|\mathrm{Q}$: By definition, $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'|\mathrm{Q} : []^b; \mathbb{C}$ and hence by the composition type rule we have $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^{b_1}; \mathbb{C}_1$ and $\Gamma_U, U : \sigma^c \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{b_2}; \mathbb{C}_2$ where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$ and $b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. By the induction hypothesis the lemma holds for P' and Q individually; i.e. $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^{b_1}; \mathbb{C}_{1U}, \mathbb{C}'_3$ and $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}\{K/U\} : []^{b_2}; \mathbb{C}_{2U}, \mathbb{C}''_3$ given that $\Gamma_U \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_3$ and $\forall V \in FV(K).(\mathbb{C}_i, \mathbb{C}_3)(V) = \mathbb{C}_1(U)$ and $\mathbb{C}_1(U) \geq b$, $i = 1, 2$ (where as required $\mathbb{C}_3 \subseteq \mathbb{C}'_3$ and $\mathbb{C}_3 \subseteq \mathbb{C}''_3$). Then by the composition type rule and the definition of substitution $\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\mathrm{P}'|\mathrm{Q})\{K/U\} : []^b; \mathbb{C}'$ where $\mathbb{C}' = \mathbb{C}_{1U}, \mathbb{C}_{2U}, \mathbb{C}'_3, \mathbb{C}''_3$ and $b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$ by the corollary to the context expansion lemma.
- $\mathrm{P} \equiv x'(\vec{V}).\mathrm{P}'$: By definition $\Gamma_{U\vec{V}}, U : \sigma^c, x : (b \cdot \vec{\sigma^d_1})^b \vdash^{\mathbb{C}_\epsilon} x(\vec{V}).\mathrm{P}' : []^e; \mathbb{C}_{1\vec{V}}, x : e$ and hence by the input type rule $\Gamma_{U\vec{V}}, U : \sigma^c, x : (b \cdot \vec{\sigma^d_1})^b, \vec{V} : b \cdot \vec{\sigma^d_1} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^e; \mathbb{C}_1$ (where $\forall V \in \vec{V}.\mathbb{C}_1(V) \geq e$). By the induction hypothesis the lemma holds for P', i.e. given $\Gamma_{U\vec{V}}, x : (b \cdot \vec{\sigma^d_1})^b \vdash^{\mathbb{C}_\epsilon} K : \sigma^c; \mathbb{C}_2$ then $\Gamma_{U\vec{V}}, x : (b \cdot \vec{\sigma^d_1})^b, \vec{V} : b \cdot \vec{\sigma^d_1} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^e; \mathbb{C}_{1U}, \mathbb{C}'_2$ with $\mathbb{C}'_2 \supseteq \mathbb{C}_2$. (note that if $U \in \vec{V}$ then lemma holds by renaming of bound variables). Now depending on whether U is a variable or a name:
  · $U \equiv x$: Then by the type rules $\Gamma_{x\vec{V}}, x : (b \cdot \vec{\sigma^d_1})^b, \vec{V} : b \cdot \vec{\sigma^d_1} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^e; \mathbb{C}_1$ and

by the definition of substitution (assuming $K \equiv y$), $\Gamma_{x\vec{V}}, y : (b \cdot \vec{\sigma_1^d})^b, \vec{V} : b \cdot \vec{\sigma_1^d} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{y/x\} : []^e; \mathbb{C}_{1\vec{V}x}, \mathbb{C}'_2$ where $\mathbb{C}'_2 \supseteq \mathbb{C}_2$ and hence $\Gamma_{x\vec{V}}, y : (b \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} y(\vec{V}).\mathrm{P}'\{y/x\} : []^e; \mathbb{C}_{1\vec{V}x}, y : e$ where $\mathbb{C}'_2 = y : e \supseteq \mathbb{C}_2 = \emptyset$ as required.

- $U \equiv X$, $K \equiv Q$: no further substitution required; i.e. assuming that $X \notin \vec{V}$ then by the type rules $\Gamma_{U\vec{V}}, x : (b \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} x(\vec{V}).\mathrm{P}'\{K/U\} : []^e; \mathbb{C}_{1U\vec{V}}, \mathbb{C}'_2$ as before.

- $\mathrm{P} \equiv \overline{x}[\vec{J}].\mathrm{P}'$: By definition, $\Gamma_U, x : (bc \cdot \vec{\sigma_1^d})^b, U : \sigma^e \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{J}].\mathrm{P}' : []^c; \mathbb{C}_1$ with $\Gamma_U, U : \sigma^e, x : (c \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} \vec{J} : c \cdot \vec{\sigma_1^d}; \mathbb{C}_2$ and hence by the output type rule $\Gamma_U, U : \sigma^e, x : (c \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^c; \mathbb{C}'_1$ (where $\mathbb{C}_1 = \mathbb{C}'_1, \mathbb{C}_2, x : c, FV\left(\vec{J}\right) : c$). By the induction hypothesis the lemma holds for P', i.e. $\Gamma_U, x : (c \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^c; \mathbb{C}'_{1U}, \mathbb{C}'_3$ given $\Gamma_U, x : (c \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} K : \sigma^e; \mathbb{C}_3$ with $\mathbb{C}_3 \subseteq \mathbb{C}'_3$. Then by the output type rule we have $\Gamma_U, x : (bc \cdot \vec{\sigma_1^d})^b \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{J}].\mathrm{P}'\{K/U\} : []^c; \mathbb{C}'_{1U}, x : c, \mathbb{C}''_3$. Similarly if $U \equiv x$ and $K \equiv y$.
- $\mathrm{P} \equiv x(V)_{\text{certify}}^{?}\mathrm{P}'{:}\mathrm{Q}$: By definition, $\Gamma_U, U : \sigma^b \vdash^{\mathbb{C}_\epsilon} x(V)_{\text{certify}}^{?}\mathrm{P}'{:}\mathrm{Q} : []^c; \mathbb{C}$ and hence by the certify type rule we have $\Gamma_U, x : (\sigma_1^d)^e, U : \sigma^b, V : \sigma_1^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^c; \mathbb{C}_1$ and $\Gamma_U, x : (\sigma_1^d)^e, U : \sigma^b, V : \sigma_1^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^c; \mathbb{C}_2$ where $\mathbb{C} = \mathbb{C}_{1V}, \mathbb{C}_{2V}, x : c$. By the induction hypothesis, lemma holds for antecedents; i.e. given $\Gamma_U, x : (\sigma_1^d)^e \vdash^{\mathbb{C}_\epsilon} K : \sigma^b; \mathbb{C}_3$ we have $\Gamma_U, x : (\sigma_1^d)^e, V : \sigma_1^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'\{K/U\} : []^c; \mathbb{C}_{1U}, \mathbb{C}'_3$ and $\Gamma_U, x : (\sigma_1^d)^e, V : \sigma_1^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}\{K/U\} : []^c; \mathbb{C}_{1U}, \mathbb{C}''_3$ where $\forall V \in FV(K).(\mathbb{C}_i, \mathbb{C}_3)(V) = \mathbb{C}(U)$ and $\mathbb{C}(U) \geq b$, $i = 1, 2$ (where as required $\mathbb{C}_3 \subseteq \mathbb{C}'_3$ and $\mathbb{C}_3 \subseteq \mathbb{C}''_3$). Hence by the certify type rule and the definition of substitution we have $\Gamma_U, x : (\sigma_1^d)^e \vdash^{\mathbb{C}_\epsilon} (x(V)_{\text{certify}}^{?}\mathrm{P}'{:}\mathrm{Q})\{K/U\} : []^c; \mathbb{C}_{1U}, \mathbb{C}_{2U}\mathbb{C}'_3, \mathbb{C}''_3, x : c$ as required.

Finally,the subject reduction theorem:

Theorem 6.9 (Subject Reduction) $\exists c \geq b, \mathbb{C}' \subseteq \mathbb{C}$.

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C} \quad \mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}}{\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^{\mathbb{R}(c)}; \mathbb{R}(\mathbb{C}')}$$

Again, there are a few subtle differences with the usual statement. The most important difference relates to the possibility of run-time coercion occurring during a reduction; this means the same assumption set cannot be used after the reduction, as the type annotations may be slightly different; to express this we include a boolean substitution — tightly defined by the reduction path — and apply that to the assumption set in the consequent.

Secondly, note that as names may expire in a reduction the context after a reduction is going to be a subset of the context before a reduction — hence by lemma 6.1 the trustedness of a process after a reduction will be at least as safe as it was before the reduction.

Proof. By induction on the derivation of $\mathrm{P} \to \mathrm{P}'$, and by cases on the structure of P.

(i) Reflex: $(\mathrm{P} \to \mathrm{P})$. Trivial.

(ii) Communication Reduction:

We proceed by considering the parts of the reduction in turn, that is $\overline{x}[\vec{K}].\mathrm{Q} \xrightarrow{\overline{x}[\vec{K}]} \mathrm{Q}'$ and $x(\vec{U}).\mathrm{R} \xrightarrow{x(\vec{K})} \mathrm{R}'\{\vec{K}/\vec{U}\}$, then considering the combination (note that subject reduction does *not* hold for the input redex in isolation; it must be considered within the context of the complete communication reduction). From the statement we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{Q}|x(\vec{U}).\mathrm{R} : []^b; \mathbb{C}$. Hence $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{Q} : []^{b_1}; \mathbb{C}_1$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} x(\vec{U}).\mathrm{R} : []^{b_2}; \mathbb{C}_2$ by the composition type rule, where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$ and $b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$.

- Firstly, consider the output case: by the output rule we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{b_1}; \mathbb{C}'_1$ where $\mathbb{C}'_1 \subseteq \mathbb{C}_1$ as follows: if $x, FV\left(\vec{K}\right) \in FV(\mathrm{Q})$ then $\mathbb{C}'_1 = \mathbb{C}_1$. Else if $x \notin FV(\mathrm{Q})$ then $\mathbb{C}'_1 = \mathbb{C}_{1x}$ and hence $\mathbb{C}'_1 \subset \mathbb{C}_1$ (and similarly $\forall y \in FV\left(\vec{K}\right)$). Therefore

$$\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{\mathbb{R}(b_1)}; \mathbb{R}(\mathbb{C}'_1)$$

  as required, where $\mathbb{R} = \mathrm{Id}$ and $b_1 \leq b_1$ by definition.
- Secondly, consider the input case: by the input type rule we have $\Gamma, \vec{U} \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^{b_2}; \mathbb{C}_2, \mathbb{C}''$ where $\mathbb{C}''$ is the subset of the names and variables $\vec{U}$ contained in $FV(\mathrm{R})$. Then by the substitution lemma (6.8) $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{R}\{\vec{K}/\vec{U}\} : []^{b_2}; \mathbb{C}_2, \mathbb{C}'''$ where $\mathrm{dom}\mathbb{C}'''$ is the subset of $FV\left(\vec{K}\right)$ substituted in R, and so $\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{R}\{\vec{K}/\vec{U}\} : []^{\mathbb{R}(b_2)}; \mathbb{R}(\mathbb{C}_2, \mathbb{C}''')$ where $\mathbb{R} = \mathrm{Id}$.

Then considering the two cases together by the composition type rule we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[\vec{K}].\mathrm{Q} : []^b; \mathbb{C}_1, \mathbb{C}_2$ and also

$$\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}|\mathrm{R}\{\vec{K}/\vec{U}\} : []^c; \mathbb{C}'_1, \mathbb{C}_2, \mathbb{C}'''$$

noting that $\mathbb{C}_1 = \mathbb{C}'_1, \mathbb{C}''', x : b$ so $\mathbb{C}'_1, \mathbb{C}''' \subseteq \mathbb{C}_1$ as required, and $c \geq b$ by lemma 6.1 (context subset).

(iii) certify Contraction:

- $\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} K : \sigma^{\mathrm{T}}$:

  Suppose the last rule used in the derivation of $\mathrm{P} \to \mathrm{P}'$ was

$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} K : \sigma^{\mathrm{T}}}{\overline{x}[K].\mathrm{R}|x(U)_{\mathrm{certify}}\mathbf{?}\mathrm{P}{:}\mathrm{Q} \to \mathrm{R}|\mathrm{P}\{K/U\}}$$

  From the statement, we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[K].\mathrm{R}|x(U)_{\mathrm{certify}}\mathbf{?}\mathrm{P}{:}\mathrm{Q} : []^c; \mathbb{C}$. Then (firstly) by the composition rule we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \overline{x}[K].\mathrm{R} : []^{c_1}; \mathbb{C}_1$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} x(U)_{\mathrm{certify}}\mathbf{?}\mathrm{P}{:}\mathrm{Q} : []^{c_2}; \mathbb{C}_2$ where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$ and $c = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$ by

definition. Then similarly by reverse applications of the output and certify rules we obtain:

$$\begin{gathered}\Gamma, x : (\sigma^b)^d \vdash^{\mathbb{C}_\epsilon} K : \sigma^b; FV(K) : c_1 \\ \Gamma, x : (\sigma^b)^d \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^{c_1}; \mathbb{C}'_1 \\ \Gamma, x : (\sigma^b)^d, U : \sigma^{\mathrm{T}} \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^{c_2}; \mathbb{C}'_2 \\ \Gamma, x : (\sigma^b)^d, U : \sigma^{\mathrm{U}} \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{c_2}; \mathbb{C}''_2\end{gathered}$$

noting that $\mathbb{C}_2 = \mathbb{C}'_2, \mathbb{C}''_2, x : c_2$ and $\mathbb{C}_1 = \mathbb{C}'_1, FV(K) : c_1, x : c_1$ (and therefore $c_1 = c_2$; we will only refer to c_1 from now on).

Now consider the antecedents (to the composition rule) in turn:

- $\overline{x}[K].\mathrm{R} \xrightarrow{\overline{x}[K]} \mathrm{R} \Rightarrow \mathbb{R}_1(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^{\mathbb{R}_1(c_1)}; \mathbb{R}_1(\mathbb{C}'_1)$ as required, where $\mathbb{R}_1 = \mathrm{Id}$ and $c_1 \leq c_1$ by definition.
- $x(U)_{\text{certify}}^{?}\mathrm{P}{:}\mathrm{Q} \xrightarrow{x(K)} \mathrm{P}\{K/U\} \Rightarrow$ $\mathbb{R}_2(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{P}\{K/U\} : []^{\mathbb{R}_2(c_1)}; \mathbb{R}_2(\mathbb{C}'_{2U}, \mathbb{C}'''_2)$ where $\mathbb{C}''' = FV(K) : c_1$ if $U \in FV(P)$, or $\emptyset$ otherwise; $c_1 \leq c_1$ by definition, and $\mathbb{R}_2 = \mathrm{Id}[b := \mathrm{T}]$.

Then composing (i.e. the composition rule):

$$\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{R}|\mathrm{P}\{K/U\} : []^{\mathbb{R}(c')}; \mathbb{R}(\mathbb{C}_1, \mathbb{C}'_2, \mathbb{C}'''_2)$$

where $\mathbb{R} = \mathbb{R}_1; \mathbb{R}_2 = \mathbb{R}_2 = \mathrm{Id}[b := \mathrm{T}]$, $\mathbb{C}_1, \mathbb{C}'_2, \mathbb{C}'''_2 \subseteq \mathbb{C} = \mathbb{C}_1, \mathbb{C}_2 = \mathbb{C}'_1, FV(K) : c_1, x : c_1, \mathbb{C}'_2, \mathbb{C}''_2$ and $c' = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x) \geq c$ by the context subset lemma (6.1) as required.

- $\Gamma \vdash^{\mathbb{C}_\epsilon}_{certify} U : \sigma^{\mathrm{U}}$: Similarly.

(iv) Sub.:

- Suppose the last rule used in the derivation was

$$\frac{\mathrm{Q} \to \mathrm{Q}'}{\mathrm{P} = !\mathrm{Q} \to !\mathrm{Q}|\mathrm{Q}' = \mathrm{P}'}$$

From the statement we have $\Gamma \vdash^{\mathbb{C}_\epsilon} !\mathrm{Q} : []^b; \mathbb{C}$, and hence by the replication type rule $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}$. Then by the induction hypothesis $\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}' : []^{\mathbb{R}(c)}; \mathbb{R}(\mathbb{C}')$ where $\mathbb{C}' \subseteq \mathbb{C}$ and $b \leq c$. Hence by the composition type rule $\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} !\mathrm{Q}|\mathrm{Q}' : []^{\mathbb{R}(b)}; \mathbb{R}(\mathbb{C})$ as required. (Note that since $\mathbb{C}' \subseteq \mathbb{C}$ then by definition $\mathbb{C}, \mathbb{C}' = \mathbb{C}$).

- Suppose the last rule used in the derivation of $\mathrm{P} \to \mathrm{P}'$ was

$$\frac{\mathrm{Q} \xrightarrow{\omega} \mathrm{Q}'}{\mathrm{P} = (\nu U)\,\mathrm{Q} \xrightarrow{\omega} (\nu U)\,\mathrm{Q}' = \mathrm{P}' \quad (U \notin FV(\omega))}$$

By the statement we have $\Gamma_U \vdash^{\mathbb{C}_\epsilon} (\nu U)\mathrm{Q} : []^b; \mathbb{C}_U$, so by the restriction type rule we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}$. Therefore by the induction hypothesis $\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}' : []^{\mathbb{R}(c)}; \mathbb{C}'$ with $\mathbb{C}' \subseteq \mathbb{C}$ and $b \leq c$, and hence by the type rules we have $\mathbb{R}(\Gamma_U) \vdash^{\mathbb{C}_\epsilon} (\nu U)\mathrm{Q}' : []^{\mathbb{R}(b)}; \mathbb{C}'_U$ where $\mathrm{Q} \to \mathrm{Q} \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}$ by definition.

- Suppose $\mathrm{P} \equiv \mathrm{Q} + \mathrm{R}$ and the last rule used in the derivation of $\mathrm{P} \to \mathrm{P}'$ was
$$\frac{\mathrm{Q} \to \mathrm{Q}'}{\mathrm{P} = \mathrm{Q} + \mathrm{R} \to \mathrm{Q}' = \mathrm{P}'}$$
From the statement we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} + \mathrm{R} : []^b; \mathbb{C}$, so by the summation type rule we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^b; \mathbb{C}_1$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^b; \mathbb{C}_2$ (where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$). Therefore by the induction hypothesis,
$$\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}' : []^{\mathbb{R}(b)}; \mathbb{R}(\mathbb{C}'_1)$$
where $\mathrm{Q} \to \mathrm{Q}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}$ by the definition of $\Vdash^{\Gamma}_{\text{certify}}$, and $\mathbb{C}'_1 \subseteq \mathbb{C}$ by definition.
- $$\frac{\mathrm{Q} \to \mathrm{Q}' \quad \mathrm{R} \to \mathrm{R}'}{\mathrm{P} = \mathrm{Q}|\mathrm{R} \to \mathrm{Q}'|\mathrm{R}' = \mathrm{P}'}$$
By the statement we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}|\mathrm{R} : []^b; \mathbb{C}$, so by the composition type rule $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^{b_1}; \mathbb{C}_1$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{R} : []^{b_2}; \mathbb{C}_2$ where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$ and $b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. Then by the induction hypothesis we have $\mathbb{R}_1(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}' : []^{\mathbb{R}_1(c_1)}; \mathbb{R}_1(\mathbb{C}'_1)$ and $\mathbb{R}_2(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{R}' : []^{\mathbb{R}_2(c_2)}; \mathbb{R}_2(\mathbb{C}'_2)$ where $b_1 \leq c_1$ and $b_2 \leq c_2$, $\mathbb{C}'_1 \subseteq \mathbb{C}_1$ and $\mathbb{C}'_2 \subseteq \mathbb{C}_2$, and $\mathrm{Q} \to \mathrm{Q}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_1$ and $\mathrm{R} \to \mathrm{R}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}_2$.
Then again by the composition rule we have
$$\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{Q}'|\mathrm{R}' : []^{\mathbb{R}(c)}; \mathbb{R}(\mathbb{C}')$$
as required, where $\mathbb{R} = \mathbb{R}_1; \mathbb{R}_2$ by definition of $\Vdash^{\Gamma}_{\text{certify}}$; $\mathbb{C}' = \mathbb{C}'_1, \mathbb{C}'_2$ and $c = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x)$, with $b \leq c$ by the context subset lemma 6.1 (since given $\mathbb{C}'_1 \subseteq \mathbb{C}_1$ and $\mathbb{C}'_2 \subseteq \mathbb{C}_2$, then $\mathbb{C}' = \mathbb{C}'_1, \mathbb{C}'_2 \subseteq \mathbb{C}_1, \mathbb{C}_2 = \mathbb{C}$).

Corollary 6.10
$\exists c \geq b$
$$\frac{\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b \quad \mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma}_{\text{certify}} \mathbb{R}}{\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^{\mathbb{R}(c)}}$$

Proof. The deduction $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b$ ended in a use of the final rule, so by lemma 6.6 (context annotation) we have $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^{b'}; \mathbb{C}$, where $b' \leq b = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x)$. Then by subject reduction (theorem 6.9) we have $\mathbb{R}(\Gamma) \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^{\mathbb{R}(c')}; \mathbb{R}(\mathbb{C}')$ where $c' \geq b'$. By lemma 6.6 again we have $c = \bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}'_T(x) \geq c'$, and by lemma 6.1 (context subset) $\mathbb{C}' \subset \mathbb{C}$ implies $c \geq b$ as required.

6.3 Correctness of the Implementation

Soundness means demonstrating that the algorithm returns a valid typing (if one exists):

Theorem 6.11 (Soundness) *If* $\langle \Gamma, \mathbb{C}, []^b \rangle = \mathrm{Type}^{\mathbb{C}_\epsilon}_\pi(\mathrm{P})$ *is defined, then* $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}$.

Proof. By induction on the structure of P.

- $\mathrm{P} \equiv \mathbf{0}$: By inspection, $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathbf{0}) = \langle \emptyset, \emptyset, []^i \rangle$ and by the zero type rule $\emptyset \vdash^{\mathbb{C}_\epsilon} \mathbf{0} : []^i; \emptyset$ as required.
- $\mathrm{P} \equiv !\mathrm{P}'$: By inspection, $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(!\mathrm{P}') = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$. By induction, if $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ then $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}$; hence by the replication type rule and the definition of $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}$ $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(!\mathrm{P}')$ and $\Gamma \vdash^{\mathbb{C}_\epsilon} !\mathrm{P}' : []^b; \mathbb{C}$ as required.
- $\mathrm{P} \equiv (\nu U)\mathrm{P}'$: By the induction hypothesis soundness holds for P'; i.e. if $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ then $\Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}$. Then by inspection of the algorithm soundness holds by the restriction type rule; $\left\langle \Gamma_U, \mathbb{C}_U, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ and $\Gamma_U \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_U$ as required.
- $\mathrm{P} \equiv \mathrm{P}'|\mathrm{Q}$: By the induction hypothesis soundness holds for P' and Q individually; i.e. $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}') = \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle \Rightarrow \Gamma_1 \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_1$ and $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{Q}) = \langle \Gamma_2, \mathbb{C}_2, []^c \rangle \Rightarrow \Gamma_2 \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^c; \mathbb{C}_2$. Then by soundness of Unify and CUnify, if $\mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2)$ and $\mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2))$ then by the composition type rule and examination of the definition $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}'|\mathrm{Q}) = \left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}), []^{\mathbb{S}_2(\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x))} \right\rangle$ and $\mathbb{S}_2(\Gamma_1 \cup \Gamma_2) \vdash^{\mathbb{C}_\epsilon} \mathrm{P}'|\mathrm{Q} : []^{\mathbb{S}_2(\bigwedge_{x \in \mathbb{C}_\epsilon} \mathbb{C}(x))}; \mathbb{S}_2(\mathbb{C})$ as required, where $\mathbb{C} = \mathbb{C}_1, \mathbb{C}_2$.
- $\mathrm{P} \equiv \mathrm{P}' + \mathrm{Q}$: Similarly, also invoking soundness of BUNIFY to unify annotations as required by the type system (summation rule).
- $\mathrm{P} \equiv x(\vec{U}).\mathrm{P}$: By the induction hypothesis soundness holds for P; i.e. $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}) = \left\langle \Gamma, \mathbb{C}, []^b \right\rangle \Rightarrow \Gamma \vdash^{\mathbb{C}_\epsilon} \mathrm{P} : []^b; \mathbb{C}$. By the soundness of Unify, $\mathbb{S}_1 = \mathrm{Unify}\Big(\Gamma, \Big\{x : (\vec{\alpha^i})^j, \vec{U} : \vec{\alpha^i}\Big\}\Big)$ ensures that the types carried by x match those of $\vec{U}$, while $\mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\Big(\mathbb{S}_1(j), \mathbb{S}_1((\vec{\alpha^i})^j)\Big)$ ensures that the input types are multiplied by the annotation of the input channel (as required by the input type rule) by the soundness of $\mathbb{M}$ and by the soundness of CUnify, $\mathbb{S}_3 = \mathbb{S}_2; \mathrm{CUnify}(\mathbb{S}_2(\mathbb{C}), \mathbb{S}_2(\{x : b\}))$ ensures that $\mathbb{C}, x : b$ can be formed (as required). Then by the type rules $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\Big(x(\vec{U}).\mathrm{P}\Big) = \Big\langle \mathbb{S}_2(\Gamma_{\vec{U}} \cup \{x : (\vec{\alpha^i})^j\}), S_2(\mathbb{C}_{\vec{U}}, x : b), []^{\mathbb{S}_2(b)} \Big\rangle$ and $\mathbb{S}_2(\Gamma_{\vec{U}} \cup \{x : (\vec{\alpha^i})^j\}) \vdash^{\mathbb{C}_\epsilon} x(\vec{U}).\mathrm{P} : []^{\mathbb{S}_2(b)}; S_2(\mathbb{C}_{\vec{U}}, x : b)$.
- $\mathrm{P} \equiv \overline{x}[\vec{K}].\mathrm{P}'$: By the induction hypothesis soundness holds for P', that is $\left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ being defined implies $\Gamma_1 \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^b; \mathbb{C}_1$. By the soundness of $*\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}$, the following also holds: $\left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma^c} \right\rangle = *\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\Big(\vec{K}\Big) \Rightarrow \Gamma_2 \vdash^{\mathbb{C}_\epsilon} \vec{K} : \vec{\sigma^c}; \mathbb{C}_2$. Then by soundness of Unify, $\mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma)$ unifies the two assumption sets, and by soundness of $\mathbb{M}$ $\mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\big(\mathbb{S}_1(b), \mathbb{S}_1(\vec{\sigma^c})\big)$ and $\mathbb{S}_3 = \mathbb{S}_2; \mathbb{M}\big(\mathbb{S}_2(i), \mathbb{S}_2((\vec{\sigma^c})^i)\big)$ ensures that the type of $\vec{K}$ is multiplied by the trustedness of P' and of x, as required. By soundness of Unify again, $\mathbb{S}_4 = \mathbb{S}_3; \mathrm{Unify}\big(\mathbb{S}_3(\Gamma_1 \cup \Gamma_2), \mathbb{S}_3(\{x : (\vec{\sigma^c})^i\})\big)$ guar-

antees that the type of x can carry names of type $\vec{\sigma^c}$ as required. By soundness of CUnify, $\mathbb{S}_5 = \mathbb{S}_4; \text{CUnify}\left(\mathbb{S}_4(\mathbb{C}_2), \mathbb{S}_4(FV\left(\vec{K}\right) : b, x : b)\right)$ and $\mathbb{S}_6 = \mathbb{S}_5; \text{CUnify}\left(\mathbb{S}_5(\mathbb{C}_1), \mathbb{S}_5(FV\left(\vec{K}\right) : b, x : b)\right)$ ensure the requirement in the output type rule that we can form $\mathbb{C}_1, x : b, \mathbb{C}_2, FV\left(\vec{K}\right) : b$ in the consequent. Then by the type rules
$\text{Type}_\pi^{\mathbb{C}_\epsilon}\left(\overline{x}[\vec{K}].\text{P}'\right) = \left\langle \mathbb{S}_5(\Gamma_1 \cup \Gamma_2) \cup \mathbb{S}_6(\{x : (\vec{\sigma^c})^i),\right.$
$\left.\mathbb{S}_5(\mathbb{C}_1 \cup \{FV\left(\vec{K}\right) : b\} \cup \{x : b\}), []^{\mathbb{S}_5(b)}\right\rangle \Rightarrow \mathbb{S}_5(\Gamma_1 \cup \Gamma_2) \cup \mathbb{S}_6(\{x : (\vec{\sigma^c})^i) \vdash^{\mathbb{C}_\epsilon}$
$\overline{x}[\vec{K}].\text{P}' : []^{\mathbb{S}_5(b)}; \mathbb{S}_5(\mathbb{C}_1 \cup \{FV\left(\vec{K}\right) : b\} \cup \{x : b\})$ as required.

- $\text{P} \equiv x(U)_{\text{certify}}^{?}\text{P}'{:}\text{Q}$: By the induction hypothesis, soundness holds for P′ and Q individually; i.e. $\text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{P}) = \left\langle \Gamma_1, \mathbb{C}_1, []^b \right\rangle \Rightarrow \Gamma_1 \vdash^{\mathbb{C}_\epsilon} \text{P} : []^b; \mathbb{C}_1$ and $\text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{Q}) = \langle \Gamma_2, \mathbb{C}_2, []^c \rangle \Rightarrow \Gamma_2 \vdash^{\mathbb{C}_\epsilon} \text{Q} : []^c; \mathbb{C}_2$. Then by the soundness of Unify the type of U in P′ (if any) is unified with fresh variable α (with annotation T and U respectively), and the assumption sets for P′ and Q are also unified together (less the type for U, as this will have a different annotation in each). Then (also by soundness of Unify) the type of x in the combined assumption sets is unified with the channel type that carries α^i (where i is a fresh variable). Lastly, by soundness of CUnify the returned contexts $\mathbb{C}_1$ and $\mathbb{C}_2$ and $x : b$ are unified, and by soundness of BUNIFY the annotations on P′ and Q are unified as required. Then by the certify type rule $\mathbb{S}_6(\Gamma_{1U}, \Gamma_{2U}, x : (\alpha^i)^j) \vdash^{\mathbb{C}_\epsilon} x(U)_{\text{certify}}^{?}\text{P}'{:}\text{Q} : []^{\mathbb{S}_6(b)}; \mathbb{S}_6(\mathbb{C}_{1U}, \mathbb{C}_{2U}, x : b)$ as required.

Correctness involves demonstrating that if a valid deduction exists for a given process, then the algorithm is defined. We additionally assert that it returns the most general typing:

Theorem 6.12 (Correctness) *If for some* $\Gamma', \mathbb{C}', \text{P}, []^c$ *there is a valid deduction* $\Gamma' \vdash^{\mathbb{C}_\epsilon} \text{P} : []^c; \mathbb{C}'$ *then* $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{P})$ *is defined, and further* $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$.

Proof. By induction on the structure of P, and by the definition of $\text{Type}_\pi^{\mathbb{C}_\epsilon}$.

- $\text{P} \equiv \mathbf{0}$: From the zero type rule (and as many applications of the weaken rule as necessary) $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathbf{0} : []^c; \emptyset$ is valid for any Γ', c. By inspection, $\text{Type}_\pi^{\mathbb{C}_\epsilon}(\mathbf{0})$ is defined and $\text{Type}_\pi^{\mathbb{C}_\epsilon}(\mathbf{0}) = \langle \emptyset, \emptyset, []^i \rangle$ (where i is a fresh variable). Then we have $\langle \emptyset, \emptyset, []^i \rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$ as $\emptyset \subseteq \Gamma'$ for any Γ' (similarly $\emptyset \subseteq \mathbb{C}'$), and $\mathbb{S}([]^i) = []^c$ where $\mathbb{S} = (\text{Id}; \text{Id}[i := c])$.
- $\text{P} \equiv !\text{P}'$: By induction, completeness holds for P′, and by inspection of the algorithm $\text{Type}_\pi^{\mathbb{C}_\epsilon}(!\text{P}') = \text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{P})$ so completeness also holds for !P′.
- $\text{P} \equiv (\nu U)\text{P}'$: By induction, completeness holds for P′: $\Gamma' \vdash^{\mathbb{C}_\epsilon} \text{P} : []^c; \mathbb{C}' \Rightarrow \left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{P})$ is defined and $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$. Then by the restriction type rule and the definition of the algorithm $\Gamma'_U \vdash^{\mathbb{C}_\epsilon} (\nu U)\text{P}' : []^c; \mathbb{C}'_U$ and $\text{Type}_\pi^{\mathbb{C}_\epsilon}((\nu U)\text{P}') = \left\langle \Gamma_U, \mathbb{C}_U, []^b \right\rangle$ with $\left\langle \Gamma_U, \mathbb{C}_U, []^b \right\rangle \leq \langle \Gamma'_U, \mathbb{C}'_U, []^c \rangle$

as required.

- $\mathrm{P} \equiv \mathrm{P}' + \mathrm{Q}$: By induction, completeness holds for P' and Q individually: $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^c; \mathbb{C}'_1 \Rightarrow \left\langle \Gamma_1, \mathbb{C}_1, []^{b_1} \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ is defined with $\left\langle \Gamma_1, \mathbb{C}_1, []^{b_1} \right\rangle \leq \langle \Gamma', \mathbb{C}'_1, []^c \rangle$, and $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathrm{Q} : []^c; \mathbb{C}'_2 \Rightarrow \left\langle \Gamma_2, \mathbb{C}_2, []^{b_2} \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{Q})$ is defined with $\left\langle \Gamma_2, \mathbb{C}_2, []^{b_2} \right\rangle \leq \langle \Gamma', \mathbb{C}'_2, []^c \rangle$. Then by the summation type rule $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' + \mathrm{Q} : []^c; \mathbb{C}'_1, \mathbb{C}'_2$. Now, by the completeness of BUNIFY $\mathbb{R} = \mathrm{BUNIFY}(b_1, b_2)$ is defined and correct, and similarly for Unify and CUnify in $\mathbb{S}_1 = \mathbb{R}; \mathrm{Unify}(\mathbb{R}(\Gamma_1), \mathbb{R}(\Gamma_2))$ and
$\mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2))$; hence
$\left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2), []^{\mathbb{S}_2(b_1)} \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P} + \mathrm{Q})$ is defined and
$\left\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2), []^{\mathbb{S}_2(b_1)} \right\rangle \leq \langle \Gamma', \mathbb{C}'_1 \cup \mathbb{C}'_2, []^c \rangle$ as required.
- $\mathrm{P} \equiv \mathrm{P}'|\mathrm{Q}$: Similarly (without the need to use BUNIFY).
- $\mathrm{P} \equiv x(\vec{U}).\mathrm{P}'$: By the induction hypothesis, completeness holds for P': $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^c; \mathbb{C}' \Rightarrow \left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ is defined and $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$. By completeness of Unify $\mathbb{S}_1 = \mathrm{Unify}\Big(\Gamma, \{x : (\vec{\alpha^i})^j, \vec{U} : \vec{\alpha^i}\}\Big)$ is defined and correct (unifying the object of the type of x with the types of the names carried by x); similarly for completeness of $\mathbb{M}$ and CUnify we have $\mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\Big(\mathbb{S}_1(j), \mathbb{S}_1((\vec{\alpha^i})^j)\Big)$ (multiplying the object of the type of x with the annotation on x as required) and $\mathbb{S}_3 = \mathbb{S}_2; \mathrm{CUnify}(\mathbb{S}_2(\mathbb{C}), \mathbb{S}_2(\{x : b\}))$ (ensuring x in the context has the same value as the trustedness of P') both defined and correct. Hence by the input type rule $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\Big(x(\vec{U}).\mathrm{P}'\Big) =$
$\Big\langle \mathbb{S}_3(\Gamma_{\vec{U}} \cup \{x : (\vec{\alpha^i})^j\}), \mathbb{S}_3(\mathbb{C}_{\vec{U}} \cup \{x : b\}), []^{\mathbb{S}_3(b)} \Big\rangle$ is defined, and
$\Big\langle \mathbb{S}_3(\Gamma_{\vec{U}} \cup \{x : (\vec{\alpha^i})^j\}), \mathbb{S}_3(\mathbb{C}_{\vec{U}} \cup \{x : b\}), []^{\mathbb{S}_3(b)} \Big\rangle \leq \langle \Gamma_{\vec{U}}, \mathbb{C}_{\vec{U}}, x : c, []^c \rangle$ as required.
- $\mathrm{P} \equiv \overline{x}[\vec{K}].\mathrm{P}'$: By the induction hypothesis, completeness holds for P': $\Gamma' \vdash^{\mathbb{C}_\epsilon} \mathrm{P}' : []^c; \mathbb{C}' \Rightarrow \left\langle \Gamma, \mathbb{C}, []^b \right\rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(\mathrm{P}')$ is defined and $\left\langle \Gamma, \mathbb{C}, []^b \right\rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$. Similarly by completeness of $*\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}$ it also holds for $\vec{K}$: $\Gamma' \vdash^{\mathbb{C}_\epsilon} \vec{K} : \vec{\sigma_1^d}; \mathbb{C}'_2 \Rightarrow \left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma^e} \right\rangle = *\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\Big(\vec{K}\Big)$ is defined and $\left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma^e} \right\rangle \leq \Big\langle \Gamma', \mathbb{C}'_2, \vec{\sigma_1^d} \Big\rangle$. Then by completeness of Unify $\mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2)$ is defined and most general, and so is $\mathbb{S}_2 = \mathbb{S}_1; \mathbb{M}\big(\mathbb{S}_1(b), \mathbb{S}_1(\vec{\sigma^c})\big)$ and $\mathbb{S}_3 = \mathbb{S}_2; \mathbb{M}\big(\mathbb{S}_2(i), \mathbb{S}_2((\vec{\sigma^c})^i)\big)$ by completeness of $\mathbb{M}$. Again by completeness of Unify, $\mathbb{S}_4 = \mathbb{S}_3; \mathrm{Unify}\big(\mathbb{S}_3(\Gamma_1 \cup \Gamma_2), \mathbb{S}_3(\{x : (\vec{\sigma^c})^i\})\big)$ is defined and by completeness of CUnify so are $\mathbb{S}_5 = \mathbb{S}_4; \mathrm{CUnify}\Big(\mathbb{S}_4(\mathbb{C}_2), \mathbb{S}_4(FV\Big(\vec{K}\Big) : b, x : b)\Big)$ and $\mathbb{S}_6 = \mathbb{S}_5; \mathrm{CUnify}\Big(\mathbb{S}_5(\mathbb{C}_1), \mathbb{S}_5(FV\Big(\vec{K}\Big) : b, x : b)\Big)$. Then by the output type rule and the definition of the algorithm $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\Big(\overline{x}[\vec{K}].\mathrm{P}'\Big)$ is defined, and $\Big\langle \mathbb{S}_5(\Gamma_1 \cup \Gamma_2) \cup \mathbb{S}_6(\{x : (\vec{\sigma^c})^i), \mathbb{S}_5(\mathbb{C}_1 \cup \{FV\Big(\vec{K}\Big) : b\} \cup \{x : b\}), []^{\mathbb{S}_5(b)} \Big\rangle \leq$

$\left\langle \Gamma', \mathbb{C}'_1, \mathbb{C}'_2, FV\left(\vec{K}\right) : c, x : c, []^c \right\rangle$ as required.

- P $\equiv x(U)_{\text{certify}}^{?}$P′:Q: By the induction hypothesis completeness holds for P′ and Q individually; i.e. $\langle \Gamma_1, \mathbb{C}_1, []^{b_1} \rangle = \text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{P}') \Rightarrow \langle \Gamma_1, \mathbb{C}_1, []^{b_1} \rangle \leq \langle \Gamma', \mathbb{C}'_1, []^c \rangle$ and $\langle \Gamma_2, \mathbb{C}_2, []^{b_2} \rangle = \text{Type}_\pi^{\mathbb{C}_\epsilon}(\text{Q}) \Rightarrow \langle \Gamma_2, \mathbb{C}_2, []^{b_2} \rangle \leq \langle \Gamma', \mathbb{C}'_2, []^c \rangle$. By completeness of Unify, CUnify, and BUNIFY the following are all defined and correct: $\mathbb{S}_1 = \text{Unify}\big(\Gamma_1, \{U : \alpha^{\text{T}}\}\big)$,
$\mathbb{S}_2 = \mathbb{S}_1; \text{Unify}\big(\mathbb{S}_1(\Gamma_2), \{U : \mathbb{S}_1(\alpha^{\text{U}})\}\big)$,
$\mathbb{S}_3 = \mathbb{S}_2; \text{Unify}(\mathbb{S}_2(\Gamma_{1U}), \mathbb{S}_2(\Gamma_{2U}))$,
$\mathbb{S}_4 = \mathbb{S}_3; \text{Unify}(\mathbb{S}_3(\Gamma_{1U} \cup \Gamma_{2U}), \mathbb{S}_3(\{x : (\alpha^i)^j\}))$,
$\mathbb{S}_5 = \mathbb{S}_4; \text{CUnify}(\mathbb{S}_4(\mathbb{C}_{1U}), \mathbb{S}_4(\mathbb{C}_{2U}))$,
$\mathbb{S}_6 = \mathbb{S}_5; \text{CUnify}(\mathbb{S}_5(\mathbb{C}_{1U}, \mathbb{C}_{2U}), \mathbb{S}_5(x : b))$,
and $\mathbb{S}_7 = \mathbb{S}_6; \text{BUNIFY}(\mathbb{S}_6(b), \mathbb{S}_6(c))$. Then by the certify type rule and the definition of the algorithm,
$\text{Type}_\pi^{\mathbb{C}_\epsilon}(x(U)_{\text{certify}}^{?}\text{P}':\text{Q}) = \left\langle \mathbb{S}_7(\Gamma_{1U}, \Gamma_{2U}, x : (\alpha^i)^j), \mathbb{S}_7(\mathbb{C}_{1U}, \mathbb{C}_{2U}, x : b_1), []^{\mathbb{S}_7(b_1)} \right\rangle$
and $\left\langle \mathbb{S}_7(\Gamma_{1U}, \Gamma_{2U}, x : (\alpha^i)^j), \mathbb{S}_7(\mathbb{C}_{1U}, \mathbb{C}_{2U}, x : b_1), []^{\mathbb{S}_7(b_1)} \right\rangle \leq \langle \Gamma', \mathbb{C}', []^c \rangle$ as required.

The soundness and correctness of the auxiliary algorithm $*\text{Type}_\pi^{\mathbb{C}_\epsilon}$ is easily proven using the previous results:

Theorem 6.13 (Soundness) *If* $\left\langle \Gamma, \mathbb{C}, \vec{\sigma^b} \right\rangle = *\text{Type}_\pi^{\mathbb{C}_\epsilon}\left(\vec{K}\right)$ *is defined, then* $\Gamma \vdash^{\mathbb{C}_\epsilon} \vec{K} : \vec{\sigma^b}; \mathbb{C}$.

Proof. By induction on the structure of $\vec{K}$, and by definition of $*\text{Type}_\pi^{\mathbb{C}_\epsilon}$.

- $\vec{K} \equiv K$: By definition, $*\text{Type}_\pi^{\mathbb{C}_\epsilon}(K) = \text{Type}_\pi^{\mathbb{C}_\epsilon}(K)$ and hence soundness holds by soundness of $\text{Type}_\pi^{\mathbb{C}_\epsilon}$.
- $\vec{K} \equiv K\vec{K'}$: By definition of the algorithm, we have $\left\langle \Gamma_1, \mathbb{C}_1, \sigma_1^b \right\rangle = \text{Type}_\pi^{\mathbb{C}_\epsilon}(K)$ and $\left\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma_2^c} \right\rangle = *\text{Type}_\pi^{\mathbb{C}_\epsilon}\left(\vec{K'}\right)$. By soundness of $\text{Type}_\pi^{\mathbb{C}_\epsilon}$ we have $\Gamma_1 \vdash^{\mathbb{C}_\epsilon} K : \sigma_1^b; \mathbb{C}_1$, and by the induction hypothesis we have $\Gamma_2 \vdash^{\mathbb{C}_\epsilon} \vec{K'} : \vec{\sigma_2^c}; \mathbb{C}_2$. By soundness of Unify $\mathbb{S}_1 = \text{Unify}(\Gamma_1, \Gamma_2)$ implies $\mathbb{S}_1(\Gamma_1 \cup \Gamma_2)$ is defined, and similarly by soundness of CUnify $\mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2)$ is defined given $\mathbb{S}_2 = \mathbb{S}_1; \text{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2))$. Hence by definition of the notation, $\mathbb{S}_1(\Gamma_1 \cup \Gamma_2) \vdash^{\mathbb{C}_\epsilon} K\vec{K'} : \sigma_1^b \vec{\sigma_2^c}; \mathbb{C}_1, \mathbb{C}_2$ as required.

Theorem 6.14 (Completeness) *If for some* $\Gamma', \mathbb{C}', \vec{K}, \vec{\sigma'^c}$ *there is a valid deduction* $\Gamma' \vdash^{\mathbb{C}_\epsilon} \vec{K} : \vec{\sigma'^c}; \mathbb{C}'$ *then* $\left\langle \Gamma, \mathbb{C}, \vec{\sigma^b} \right\rangle = *\text{Type}_\pi^{\mathbb{C}_\epsilon}\left(\vec{K}\right)$ *is defined, and further* $\left\langle \Gamma, \mathbb{C}, \vec{\sigma^b} \right\rangle \leq \left\langle \Gamma', \mathbb{C}', \vec{\sigma'^c} \right\rangle$.

Proof. By induction on the structure of $\vec{K}$, and by definition of $\text{Type}_\pi^{\mathbb{C}_\epsilon}$.

- $\vec{K} \equiv K$: By definition, $*\text{Type}_\pi^{\mathbb{C}_\epsilon}(K) = \text{Type}_\pi^{\mathbb{C}_\epsilon}(K)$ and hence completeness holds by completeness of $\text{Type}_\pi^{\mathbb{C}_\epsilon}$.
- $\vec{K} \equiv K\vec{K'}$: $\Gamma' \vdash^{\mathbb{C}_\epsilon} K\vec{K'} : \sigma_1'^d \vec{\sigma_2'^e}; \mathbb{C}'_1, \mathbb{C}'_2$ implies $\Gamma' \vdash^{\mathbb{C}_\epsilon} K : \sigma_1'^d; \mathbb{C}'_1$ and

$\Gamma' \vdash^{\mathbb{C}_\epsilon} \vec{K'} : \vec{\sigma_2'^e}; \mathbb{C}_2'$. By definition of the algorithm we have $\langle \Gamma_1, \mathbb{C}_1, \sigma_1^b \rangle = \mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(K)$ and $\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma_2^c} \rangle = *\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\left(\vec{K'}\right)$. By completeness of $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}$ $\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}(K)$ is defined with $\langle \Gamma_1, \mathbb{C}_1, \sigma_1^b \rangle \leq \langle \Gamma', \mathbb{C}_1', \sigma_1'^d \rangle$, and by the induction hypothesis $*\mathrm{Type}_\pi^{\mathbb{C}_\epsilon}\left(\vec{K'}\right)$ is also defined with $\langle \Gamma_2, \mathbb{C}_2, \vec{\sigma_2^c} \rangle \leq \left\langle \Gamma', \mathbb{C}_2', \vec{\sigma_2'^e} \right\rangle$. Then by completeness of Unify and CUnify respectively, both $\mathbb{S}_1 = \mathrm{Unify}(\Gamma_1, \Gamma_2)$ and $\mathbb{S}_2 = \mathbb{S}_1; \mathrm{CUnify}(\mathbb{S}_1(\mathbb{C}_1), \mathbb{S}_1(\mathbb{C}_2))$ are defined and most general. Hence by definition of the notation $\langle \mathbb{S}_2(\Gamma_1 \cup \Gamma_2), \mathbb{S}_2(\mathbb{C}_1 \cup \mathbb{C}_2), \mathbb{S}_2(\sigma_1^b \vec{\sigma_2^c}) \rangle \leq \left\langle \Gamma', \mathbb{C}_1', \mathbb{C}_2', \sigma_1'^d \vec{\sigma_2'^e} \right\rangle$ as required.

7 Conclusions

An annotated type system for a higher-order π-calculus was presented. In addition to enforcing regular safety properties, this was able to prevent insecure data from being used in a trusted calculation. To extend the power of the system, we introduced a simple syntax extension to the language that enabled statically reasoning about (and proving the safety of) run-time coercion of data trustedness.

To reason about and prove properties of the difficulties raised by a higher-order system, including the ability to assign processes different trust values in different environments depending on their ability to interact with the environment, the concept of execution contexts was introduced.

A statement of subject reduction, suitably altered to reflect the dynamic nature of the type system, was described and proven.

A type inference algorithm was presented, and statements of soundness and correctness proven.

7.1 Related Work

The higher-order calculus and its type system were based on that presented by Vasconcelos [20].

The prime inspiration for this research is the work of Ørbæk and Palsberg [13,14]. They described systems of both a functional and imperative nature, and introduced the notion of run-time coercions on trust (that were still able to be statically proven safe, modulo correct usage of the coercion operators). Ours differs from theirs by implementing the results in a distributed system, and by making all coercion implicit in the language thereby removing the responsibility of correctly applying it from the programmer. We also provide a modular (abstract) certification framework.

The capabilities provided by the base annotated type system (minus coercion) reflect those of many systems, commonly referred to as flow analysis [19,21,3,15,5]. To the best of our knowledge, however, none of these provide the ability to safely coerce security information at run-time.

7.2 Future Extensions

The system presented is a solid framework, however there are many useful programs that cannot currently be typed due to a lack of flexibility in either the underlying type system or the handling or trustedness information. Two proposed extensions would admit a larger set of programs: recursive types, and a form of subtyping.

Recursive types are already present in most base type systems for the π-calculus [20], however since they are not vital they were omitted here as they complicate the inference algorithm, and care must be taken in the handling of annotations.

A subtyping relation would also allow many more programs to be typed; in particular the form currently forbidden by the form of the output rule, as demonstrated in example 4.5.

Acknowledgement

The authors are grateful to Adam Berry, Julian Dermoudy and Chris Mitchell for their proof reading and comments. Hepburn was partly supported in this research by a Tasmanian Postgraduate Research Scholarship.

References

[1] Abadi, *Security protocols and specifications*, in: *FOSSACS: International Conference on Foundations of Software Science and Computation Structures* (1999).

[2] Abadi, M., C. Fournet and G. Gonthier, *Secure implementation of channel abstractions*, in: *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998, pp. 105–116.

[3] Denning, D. E., *A lattice model of information flow*, Communications of the ACM **19** (1976), pp. 236–243.

[4] Hepburn, M. and D. Wright, *Trust in the pi-calculus*, in: *Third International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, 2001.

[5] Honda, K., V. Vasconcelos and N. Yoshida, *Secure information flow as typed process behaviour*, in: G. Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2000), (Berlin, Germany, March/April 2000)*, lncs **1782** (2000), pp. 180–199.

[6] Huet, G., *Formal structures for computation and deduction* (1986), first Edition.

[7] Martin, U. and T. Nipkow, *Boolean unification — the story so far*, in: C. Kirchner, editor, *Unification*, Academic Press, 1990 pp. 437–456.

[8] Milner, R., "A Calculus for Communicating Systems," Lecture Notes in Computer Science **92**, Springer-Verlag, Berlin, 1980.

[9] Milner, R., *The polyadic π-calculus: A tutorial*, in: F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Springer-Verlag, 1993 .

[10] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, I*, Information and Computation **100** (1992), pp. 1–40.

[11] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, II*, Information and Computation **100** (1992), pp. 41–77.

[12] Necula, G. C. and P. Lee, *Safe, untrusted agents using proof-carrying code*, in: G. Vigna, editor, *Mobile Agents and Security*, Lecture Notes in Computer Science **1419** (1998).

[13] Ørbæk, P., *Can you Trust your Data?*, in: M. I. S. P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, Springer Lecture Notes in Computer Science **915 of LNCS** (1995), pp. 575–590.

[14] Palsberg, J. and P. Ørbæk, *Trust in the λ-calculus*, in: A. Mycroft, editor, *SAS'95: Static Analysis*, Lecture Notes in Computer Science **983** (1995), pp. 314–330.

[15] Pottier, F. and S. Conchon, *Information flow inference for free*, in: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montral, Canada, 2000, pp. 46–57.
URL `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`

[16] Sangiorgi, D., *From π-calculus to Higher-Order π-calculus — and back*, in: M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93*, Lecture Notes in Computer Science **668** (1993), pp. 151–166.

[17] Sewell, P. and J. Vitek, *Secure composition of insecure components*, Trusted objects, Centre Universitaire d'Informatique, University of Geneva (1999).

[18] Sewell, P. and J. Vitek, *Secure composition of untrusted code: Wrappers and causality types*, Technical Report 478, Computer Laboratory, University of Cambridge (1999).

[19] Smith, G. and D. Volpano, *Secure information flow in a multi-threaded imperative language*, in: ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium, San Diego, California, 19–21 January 1998* (1998), pp. 355–364.

[20] Vasconcelos, V. T., *A note on a typing system for the higher-order π-calculus* (1993).

[21] Volpano, D., G. Smith and C. Irvine, *A sound type system for secure flow analysis*, Journal of Computer Security **4** (1996), pp. 167–187.

[22] Wright, D. A., "Reduction Types and Intensionality in the Lambda-Calculus," Ph.D. thesis, University of Tasmania (1992).

[23] Yahalom, R., B. Klein and T. Beth, *Trust relationships in secure systems—a distributed authentication perspective* (1993).

[24] Yahalom, R., B. Klein and T. Beth, *Trust-based navigation in distributed systems* (1994).

[25] Yellin, F., *Low level security in Java*, in: *Fourth International Conference on the World-Wide Web*, MIT, Boston, 1995.
URL `http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html`